Conception de SoC à Base d'Horloges Abstraites : Vers l'Exploration d'Architectures en MARTE

Adolf Samir Abdallah

Conception de SoC à Base d'Horloges Abstraites : Vers l'Exploration d'Architectures en MARTE

Thèse présentée par

Adolf Samir ABDALLAH

En vue de l'obtention du grade de

docteur (Ph.D.)

en Informatique

À

L'École Doctorale Sciences pour l'Ingénieur

UNIVERSITÉ DES SCIENCES ET TECHNOLOGIES DE LILLE - FRANCE

Membre de Jury:

Mohamed ABID	Professeur ENIS	**Examinateur**
Frédéric ROBERT	Professeur Université Libre de Bruxelles	**Rapporteur**
Jean-Philippe BABAU	Professeur Université de Bretagne Occidentale	**Rapporteur**
Jean-Luc DEKEYSER	Professeur Université de Lille 1	**Directeur**
Abdoulaye GAMATIÉ	Chercheur CNRS	**Co-directeur**

Université des Sciences et Technologies de Lille
LIFL - UMR 8022 - Cité Scientifique, Bât. M3 - 59655 Villeneuve d'Ascq Cedex

Dédicace

Aux âmes de mes grands pères, mon oncle Max et ma grand mère

À mon cher père Samir et ma chère mère Colette
À ma sœur Rita et mon frère Christian
À ma grand mère Violette, Fadwa, Rita et Bella
Pour votre extrême amour, vos prières, vos aides précieuses et vos encouragements depuis de nombreuses années
Que Dieu vous préserve bonne santé et longue vie

À mes oncles Body et Maroun
Merci pour vos encouragements
Je vous souhaite tout le bonheur du monde, la réussite et la prospérité
Que Dieu vous garde

À mes tantes Marina, Annette, Rosette
Je vous souhaite une longue vie pleine de bonheur
Que Dieu vous garde

À tous mes amis

Remerciements

C'est avec une grande émotion et beaucoup de sincérité que je souhaiterais exprimer ma profonde reconnaissance à tous ceux qui ont permis de mener à bien ce travail.

Cette thèse n'aurait vu le jour sans la confiance, la patience et la générosité de mon directeur de recherche, Professeur Jean-Luc DEKEYSER, que je veux vivement remercier. Recevez, Monsieur DEKEYSER, mes plus sincères remerciements pour m'avoir accueilli dans votre laboratoire. Je tiens également à vous exprimer toute ma reconnaissance pour ces trois années de thèse que j'ai passées à vos côtés. Au cours de ces années, votre grande disponibilité, votre rigueur scientifique, votre enthousiasme et vos précieux conseils m'ont permis de travailler dans les meilleures conditions.

Je tiens à remercier aussi Monsieur Abdoulaye GAMATIÉ, mon encadrant de thèse. Vous avez su me laisser la liberté nécessaire à l'accomplissement de mes travaux, tout en y gardant un œil critique et avisé. Nos continuelles discussions ainsi que vos conseils efficaces, orientations et remarques pertinentes ont sûrement été la clé de succès de cette thèse. Plus qu'un encadrant ou un collègue, je crois avoir trouvé en vous un ami qui m'a aidé aussi bien dans le travail que dans la vie lorsque j'en avais besoin.

Mes plus sincères remerciements vont aux membres du jury qui ont accepté d'évaluer mon travail de thèse. Merci à Professeur Mohamed ABID, pour l'honneur qu'il m'a fait en acceptant de présider le jury de cette thèse. Mes remerciements les plus respectueux vont à Professeur Jean-Philippe BABAU et Professeur Frédéric ROBERT qui m'ont fait l'honneur de prendre connaissance de ce travail et d'en être rapporteur. Qu'ils trouvent ici l'expression de ma profonde reconnaissance.

Un grand remerciement est adressé à tous mes collègues, membres de l'équipe DaRT pour l'ambiance très favorable qu'ils ont su créer autour de moi. Enfin, mes remerciements vont également à tous ceux qui ont participé plus ou moins indirectement au bon déroulement de ma thèse.

TABLE DES MATIÈRES

Chapter 1

Introduction

1.1 Conception de systèmes de traitement intensif de données

Les applications contenant des traitements intensifs de données sont devenues courantes dans plusieurs domaines parmi lesquels, le multimédia (vidéoconférence, consoles de jeux vidéos), la défense et sécurité (reconnaissance d'objets 3D, radars, sonars, GPS), le médical (imagerie médicale pour la chirurgie assistée par ordinateur).

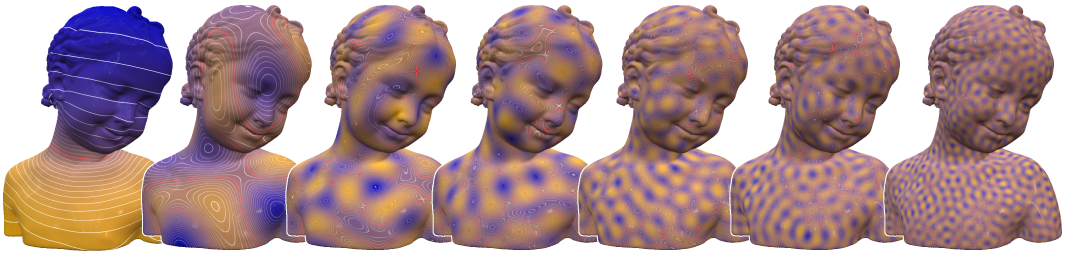


Figure 1.1: Application de la transformée de Fourrier sur un maillage [125].

Par exemple, la Figure 1.1 montre une application haute performance pour analyser et reconnaître un visage (ou maillage). La transformée de Fourier est appliquée sur l'ensemble des sommets du maillage. Ce calcul est nécessaire afin de trouver les vecteurs propres de chaque sommet selon l'opérateur *Laplace Beltrami*. Quand le contenu du maillage dépasse quelques milliers de sommets, une grande puissance de calcul est nécessaire afin de traiter et de stocker ces vecteurs [125].

La conception de systèmes embarqués sur puces, pour une partie de ces applications, présente de nombreux avantages : petite taille, faible consommation d'énergie et performance de calculs raisonnable. On parle de *systèmes-sur-puce*, en anglais System-on-Chip (SoC). Néanmoins, les technologies de conceptions adaptées pour ces SoC ne sont pas triviales. En effet, on a besoin d'aller au delà des limitations en puissance de calculs et en consommation d'énergie induites par certaines plateformes, notamment les architectures matérielles mono-processeur. L'augmentation des fréquences du processeur dans ces

architectures ne permet pas d'assurer le besoin croissant de puissance de calculs pour ces applications.

Actuellement, plusieurs processeurs peuvent être intégrés sur une même puce. Ce genre de SoC est qualifié de MPSoC pour *Multi-Processor SoC*. Par exemple, un MPSoC peut être composé de plusieurs types de processeurs tels que les microprocesseurs, les DSP (Digital Signal Processor) et les accélérateurs matériels; amenant à la conception des systèmes multiprocesseur hétérogènes sur puce (MPSoC). Cette richesse dans le nombre et le type de processeurs augmente significativement la puissance de calcul et améliore les performances. Cela permet d'implémenter des applications complexes tout en conservant un haut niveau de fiabilité et de sécurité.

Par ailleurs, on assiste à une explosion du nombre de fonctionnalités dans les systèmes embarqués à l'image des dernières générations de téléphones cellulaires. Ces appareils incluent une riche palette d'applications : musique, vidéo, Internet, GPS, téléphonie, photographie. Cela induit une complexité supplémentaire dans leur développement. Malheureusement, les approches existantes [58] pour la conception de ces systèmes sur des MPSoC sont loin d'être satisfaisantes. Par exemple, celles basées sur le niveau d'abstraction appelé *Register Transfer Level* (RTL) [24, 109], abordant les circuits en terme de transferts de données, requièrent un niveau de détails très fin qui rend l'activité de conception à la fois longue en durée et fastidieuse en complexité. Cela entrave significativement la productivité des développeurs nécessaire à la maîtrise des contraintes de mise à disposition sur le marché. Les approches visant des niveaux d'abstractions plus élevés pour les MPSoC [134, 123] sont seulement en cours d'émergence. De plus, aucune parmi elles traite le parallélisme potentiel existant dans les applications hautes performances.

Les observations que nous venons de faire ci-dessus posent un défi méthodologique auquel nous nous intéressons dans cette thèse. Pour ce fait, il est primordial de disposer des ingrédients suivants :

- un modèle de calcul : capacité à exprimer le parallélisme potentiel dans le système entier (algorithmique, architecture matérielle et association) ;
- des techniques d'analyse de choix de conception : capacité à évaluer rapidement différents choix ;
- un cadre pratique de conception : mise à disposition des utilisateurs un environnement de conception et d'analyse de MPSoC pour des applications nécessitant du calcul haute performance ;

Bref, en raison de la complexité et de l'hétérogénéité de MPSoC, il est nécessaire de mettre en place des méthodes et des outils facilitant la conception de ces systèmes.

1.2 Problématique

Une exploitation intelligente du parallélisme potentiel des ressources physiques permet d'avoir des exécutions plus performantes du système. Cependant, la conception demeure de plus en plus complexe nécessitant beaucoup de temps. Cela est dû principalement au grand nombre de ressources physiques à gérer et à la grande quantité de données à traiter dans le cas des applications hautes performances. D'où la nécessité d'un modèle de calcul efficace et facile à programmer. La première question à laquelle nous allons répondre est la suivante :

Q1 : Quel modèle de calcul est adapté pour décrire de façon compacte, le parallélisme de tâches et de données et les architectures massivement parallèles ?

Dans le but de maîtriser la complexité dans la conception et de répondre aux contraintes de temps de mise sur le marché et de coût, la conception des MPSoC doit se faire dans plusieurs niveaux d'abstraction, en commençant par le niveau le plus élevé, afin de faciliter le travail au concepteur. Néanmoins, l'élévation du niveau d'abstraction doit être accompagnée par des processus de raffinement. Ces processus offrent une passerelle vers des mises en œuvre effectives. Cela est dû au "*gap*" qui existe entre d'une part la spécification à un haut niveau d'abstraction du système et d'autre part son implémentation réelle. À chaque étape de raffinement d'un niveau d'abstraction à un autre, des détails supplémentaires doivent être insérés. Cela réduit le fossé existant entre la spécification et la réalisation physique. La problématique est alors :

Q2 : Quel flot de conception est adéquat pour le développement des MPSoC ?

Il est préférable d'éviter les simulations de système à un bas niveau d'abstraction dans les débuts du processus de conception. En effet, le temps et l'effort de la simulation des algorithmes, d'une part, et la précision des résultats de simulation, d'autre part, dépendent énormément du niveau d'abstraction auquel les modèles d'algorithmes et d'architecture sont décrits. Au niveau RTL par exemple, le temps de simulation est très long alors que les résultats sont très précis. Au niveau système, la simulation demeure plus souple et rapide (un facteur de 400 fois voir plus selon la taille de l'application) en offrant une infrastructure flexible [21]. Cela est dû à l'abstraction de certains détails et techniques d'implémentation. Ceci nous amène à poser la question suivante :

Q3 : Comment analyser rapidement des propriétés fonctionnelles (dépendances de données) et non fonctionnelles (temps, énergie) d'applications implantées sur MPSoC ?

Le concepteur peut simuler plusieurs fois différents modèles d'algorithmes (type, taille et distribution de données manipulées, etc.) et différents modèles d'architectures (taille des mémoires, nombre et valeur de fréquence des processeurs, etc.). Pour chaque simulation, le concepteur vérifie si les résultats obtenus convergent vers les résultats souhaités. Parmi les nombreuses combinaisons possibles, seul un petit nombre sera acceptable. Ceci représente un travail répétitif, d'une grande difficulté et consommant beaucoup de temps. Pour cela, l'étape d'analyse et de vérification doit être automatisée afin de pouvoir tester rapidement les différentes configurations possibles du système. Cela entraîne l'interrogation suivante :

Q4 : Comment mettre en œuvre les techniques d'analyses retenues sur le modèle de calcul identifié ?

La section suivante esquisse les réponses proposées par cette thèse concernant chacune des questions relevées ci-dessus.

1.3 Contribution de la thèse

Les deux lignes directrices suivies au cours de ce travail sont l'analyse et la vérification de l'espace de conception.

Nous nous basons sur l'approche de conception conjointe (en anglais *co-design*) d'applications et d'architectures pour faire face à la complexité croissante de MPSoC. Nous adoptons le profil Marte [89] dédié à la modélisation conjointe des systèmes embarqués temps réel. Ce profil contient un formalisme graphique permettant de représenter, de manière compacte, des applications ayant des traitements intensifs et réguliers de données ainsi que des architectures massivement parallèles.

Nous proposons une méthodologie d'analyse et de vérification d'algorithmes et d'architectures au niveau système afin d'éviter les problèmes de conception décrits dans la section précédente.

Cette méthodologie doit avoir conscience de certaines caractéristiques clés des applications et des architectures visées. De cette façon, plusieurs configurations d'algorithmes et d'architectures peuvent être explorées plus rapidement en comparaison avec des analyses faites à des niveaux plus bas, par exemple TLM ou RTL. Cela est réalisé tout en respectant des contraintes temporelles imposées par le concepteur.

Le travail réalisé se compose de trois axes, décrits ci-dessous.

Abstraction de comportements de MPSoC à l'aide d'horloges abstraites

Nous adoptons un flot de conception en Y, permettant de modéliser conjointement, à l'aide du profil Marte, des applications hautes performances sur des architectures multiprocesseur massivement parallèles. Le modèle Marte, résultant de cette modélisation conjointe, est décrit à l'aide d'horloges abstraites dont la notation est inspirée des horloges k-périodiques [29].

D'abord, nous définissons des horloges abstraites fonctionnelles pour décrire les dépendances de tâches au niveau fonctionnel. Une application, décrite en Marte, contient un niveau de parallélisme de tâches très puissant. Pour cela, nous proposons d'abstraire le modèle fonctionnel selon un des trois types de modèle suivants : *parallèle*, *pipeline* ou *mixte*. Ces derniers définissent le niveau de parallélisme que nous souhaitons capter dans les horloges abstraites. Dans le niveau *parallèle*, nous exploitons au maximum le parallélisme de tâches sous-jacent dans une application décrite en Marte. Nous considérons que toutes les tâches élémentaires sont exécutées en parallèle sur des processeurs virtuels. Dans l'exécution *pipeline*, nous restreignons le niveau parallèle à un niveau moins performant en nombre de processeurs virtuels. Dans l'exécution *mixte*, une exécution *parallèle* est considérée pour certaines fonctionnalités alors qu'une exécution *pipeline* est considérée pour d'autre.

Ensuite, nous définissons des horloges abstraites physiques décrivant la vitesse d'exécution des différents processeurs de l'architecture considérée. Les horloges abstraites fonctionnelles sont ensuite projetées sur les horloges abstraites physiques. Cette projection se base sur un algorithme d'ordonnancement, statique non préemptif, de tâches sur des processeurs. Le résultat de cette projection est une nouvelle trace d'horloges abstraites d'exécution représentant l'activité des différents processeurs durant l'exécution des fonctionnalités. Cette trace nous sera utile pour vérifier des contraintes fonctionnelles et non fonctionnelles du système.

Analyse de MPSoC basée sur les horloges abstraites

Il n'est pas évident d'estimer manuellement le meilleur choix d'association entre une application et une architecture, surtout pour des applications hautes performances et des architectures massivement parallèles. Un des objectifs de cette thèse est de vérifier des propriétés fonctionnelles et non fonctionnelles d'applications hautes performances et ciblant des architectures multiprocesseur. Cette thèse traite cette limite en présentant une méthodologie d'analyse de MPSoC, à base d'horloges abstraites et à un haut niveau d'abstraction. Cette méthodologie traite la correction fonctionnelle du système, en analysant des contraintes de dépendance de tâches, ainsi que certaines propriétés non fonctionnelles tels que les performances et la consommation d'énergie de systèmes.

L'analyse que nous proposons considère certains critères : le type, le nombre et les fréquences des processeurs impliqués dans l'exécution ainsi que les choix d'associations de tâches à des processeurs. La modification d'un de ces paramètres affecte directement le fonctionnement du système tel que : le temps d'exécution total de l'application, la consommation d'énergie, la correction fonctionnelle.

L'objectif final de la méthodologie proposée est de pousser le concepteur de MPSoC à éliminer, dès le départ dans le flot de conception, un grand nombre de mauvaises configurations.

Implémentation de la méthodologie proposée

Nous proposons une mise en œuvre de la méthodologie présentée ci-dessus en générant automatiquement, à l'aide des techniques de transformation de modèles, des horloges abstraites fonctionnelles et physiques. Pour cela, nous définissons un métamodèle intermédiaire d'horloges. Ce dernier représente le métamodèle Marte mais avec des notions d'horloges abstraites fonctionnelles et physiques. Une transformation de modèle, écrite en QVTO, permet de générer automatiquement un modèle conforme au métamodèle intermédiaire, à partir d'un modèle conforme au métamodèle Marte. Une trace d'horloges abstraites résulte de cette transformation. Cette dernière est ensuite analysée par inspection à l'aide d'un outil de vérification basé sur des fonctions écrites en OCaml.

Finalement, la méthodologie d'analyse et de vérification à haut niveau, que nous proposons dans cette thèse, peut être adaptée à tout environnement basé sur la conception conjointe de MPSoC.

1.4 Organisation de la thèse

Cette thèse est divisée en trois parties :

1. dans la première partie, nous introduisons les notions de base concernant la conception de MPSoC à différents niveaux d'abstraction (chapitre 2). Nous présentons ensuite un état de l'art sur les techniques existantes dans la littérature abordant l'analyse de MPSoC (chapitre 3). Enfin, nous présentons l'environnement Gaspard2, dédié à la conception et l'analyse de MPSoC hautes performances (chapitre 4) ;

2. dans la deuxième partie, nous modélisons de comportements fonctionnels contraints par le biais d'horloges abstraites fonctionnelles (chapitre 5). Ensuite, nous modélisons des architectures massivement parallèles par le biais d'horloges abstraites physiques. Nous proposons un ensemble d'algorithmes permettant la projection d'horloges fonctionnelles sur des horloges physiques (chapitre 6). Le

résultat de cette projection est un ensemble d'horloges simulant l'exécution de fonctionnalités. Dans le chapitre suivant (chapitre 7), nous proposons des techniques d'analyses de propriétés fonctionnelles et non fonctionnelles, en considérant les horloges abstraites comme support d'analyse ;

3. la dernière partie est une validation du modèle d'horloges abstraites proposé, et des algorithmes exposés dans la partie précédente. En particulier, nous décrivons dans le chapitre 8 un noyau de manipulation d'horloges pour une intégration dans l'environnement Gaspard2. Nous présentons dans le chapitre 9 des résultats expérimentaux sur une étude de cas consistant d'un encodeur JPEG implanté sur plusieurs configurations d'architectures multiprocesseur. Cette étude de cas permet, non seulement de donner un aperçu de la méthodologie d'analyse de modèles dans Gaspard2, mais aussi de valider les différentes propositions faites précédemment.

Enfin, nous concluons et nous formulons quelques perspectives concernant l'ensemble de nos travaux.

Part I

Approches de conception et d'analyse de MPSoC

Présentation générale

Dans cette partie nous nous intéresserons aux méthodologies de conception et aux techniques d'analyses de MPSoC destinées aux traitements intensifs de données. La motivation principale de notre travail est la mise au point d'une méthodologie permettant de restreindre, très tôt dans le flot de conception, l'espace de solutions possibles de configurations. Cela permet de minimiser l'effort de conception et de respecter le temps de la mise à disposition de produits électroniques sur le marché.

En effet, l'une des difficultés rencontrées par les concepteurs de ces systèmes est de trouver les meilleures configurations de fonctionnalités, d'architectures et d'associations de ces deux dernières. L'exploration vise principalement à trouver la meilleure configuration d'un système tout en respectant les contraintes imposées sur l'application. Cet objectif doit être atteint tout en garantissant un temps de conception et de test relativement court pour évaluer un nombre important d'alternatives.

Dans ce contexte, nous divisons l'état de l'art en 3 chapitres :

- dans le chapitre 2, nous définirons d'abord les caractéristiques des MPSoC et leurs domaines d'applications. Ensuite, nous présenterons des méthodologies de conceptions de MPSoC dans différents niveaux d'abstraction ;

- dans le chapitre 3, nous présenterons un état de l'art sur des propriétés non fonctionnelles de MPSoC (temps d'exécution, énergie dissipée, niveau de parallélisme de tâches et de données et hiérarchie de mémoires). Nous définirons d'abord ces propriétés et ensuite, nous explorons des travaux existants dans la littérature qui exploitent ces caractéristiques ;

- dans le chapitre 4, nous présenterons Gaspard2, un environnement de co-modélisation de MPSoC dédié au traitement d' applications hautes performances. Nous nous appuierons sur cet environnement pour présenter l'analyse de systèmes à base d'horloges abstraites, que nous proposons dans cette thèse.

Chapter 2

Conception des MPSoC

2.1 Introduction

Dans le chapitre 1, nous avons montré dans un premier temps la problématique rencontrée lors de la conception des systèmes électroniques appelés MPSoC. Cette problématique est une conséquence directe de l'avancement de la technologie des MPSoC qui doivent à leurs tours répondre à des demandes de plus en plus strictes par rapport aux différentes exigences d'applications hautes performances. Dans un second temps, nous avons présenté les différentes contributions que nous avons apportées dans ce travail.

L'objectif de ce chapitre est de présenter d'abord une introduction générale des systèmes MPSoC. La suite de ce chapitre étale les différentes méthodologies de conception de MPSoC. Le chapitre se poursuivra par un état de l'art détaillé sur des environnements de conception de MPSoC.

2.2 Présentation générale

Les appareils numériques de notre vie quotidienne, dont une partie est connue sous le nom de systèmes sur puce, doivent, d'une part avoir une grande vitesse de calcul quand des applications hautes performances sont considérées et, d'autre part, consommer peu d'énergie pour allonger le plus possible leur autonomie. Durant la phase de conception, la précision des résultats de simulation de ces systèmes dépend énormément du niveau

d'abstraction choisi. Ce critère représente un aspect critique pour le concepteur afin d'assurer la qualité et la correction des résultats et des mesures.

Dans le cadre de cette thèse, le terme "conception au niveau système" est attribué à tout niveau d'abstraction qui est au-dessus du niveau transfert de registre.

Le niveau algorithmique (décrivant les fonctionnalités de l'application) et celui transactionnel (décrivant des mécanismes de communication à travers des FIFO) sont deux niveaux possibles pour la conception de MPSoC au niveau système.

2.2.1 Qu'est-ce qu'un MPSoC ?

Dans cette section, nous allons définir l'architecture MPSoC.

Definition 1 (SoC) *Un SoC représente un circuit électronique intégré sur une seule puce, pouvant comprendre un processeur, des mémoires, des périphériques d'interface, et tout autre composant nécessaire à la réalisation de fonctions attendues [131].*

La Figure 2.1 montre une description d'un SoC typique fabriqué par `Freescale Semiconductor` [47] contenant un processeur tournant à une fréquence allant jusqu'à 32 MHz et une fréquence de bus allant jusqu'à 16 MHz. Ce SoC est dédié au traitement de nombreuses applications dans différents domaines tels que le domestique (télécommande télévision, clés et serrures électroniques, manette sans fil), la santé (détecteur de gaz et de vibration, machine de surveillance du patient) et dans l'industrie (surveillance d'incendie, contrôleur d'accès, contrôleur de chauffe-eau, chauffage et ventilation).

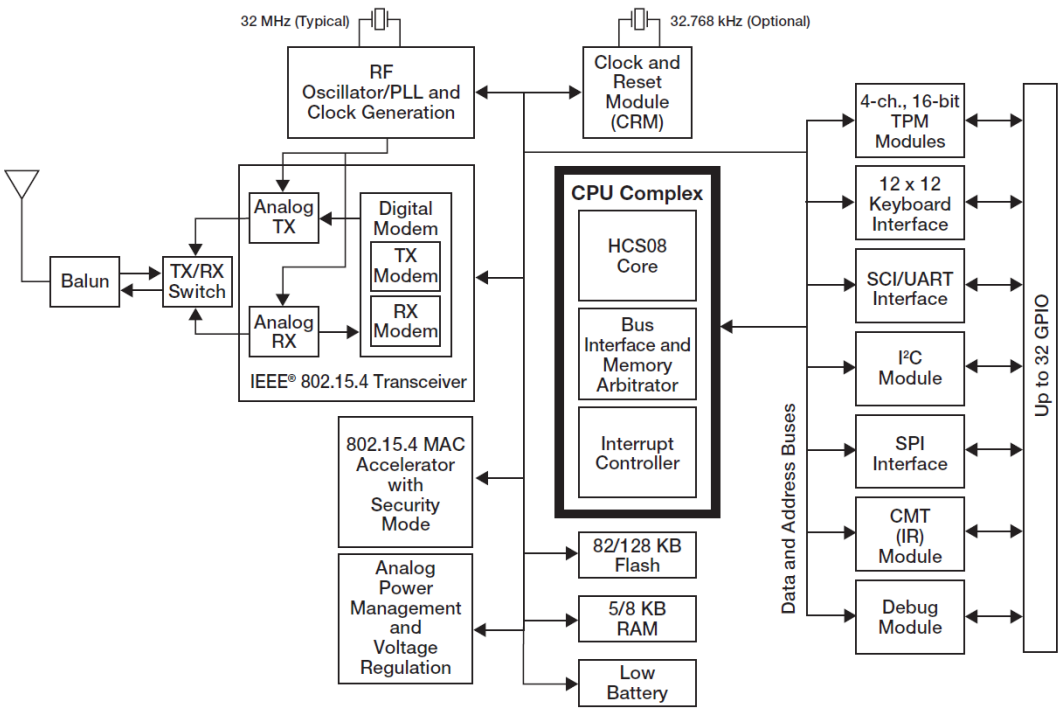


Figure 2.1: Exemple d'un SoC fabriqué par `Freescale Semiconductors` et contenant un CPU et des interfaces de communications et de stockage [46].

La conception des SoC ne cesse pas de s'accroître. L'approche de base est d'intégrer de plus en plus de fonctionnalités dans un même système électronique. Cette intégration

peut contenir du matériel ou du logiciel. Pour cela, les MPSoC ont été proposés comme une réponse à la limitation en puissance de calcul des SoC mono-processeur. En effet, la nécessité de plus de puissance de calcul pour le traitement d'applications hautes performances ne pourra pas être toujours suivie par une augmentation de la fréquence des processeurs. L'augmentation de la fréquence induit non seulement une augmentation de la puissance de calcul mais aussi de la consommation d'énergie.

Definition 2 (MPSoC) *Un MPSoC représente un système multiprocesseur embarqué sur une seule puce, conçu pour satisfaire les exigences d'une application embarquée [133]. Il se compose habituellement d'un ou de plusieurs processeurs de nature différente (des microprocesseurs, des processeurs de traitement de signal (DSP), des micro-contrôleurs), de mémoires et de quelques périphériques tels qu'un contrôleur d'accès mémoire (DMA) et un contrôleur d'entrées/sorties; le tout étant intégré sur une même puce.*

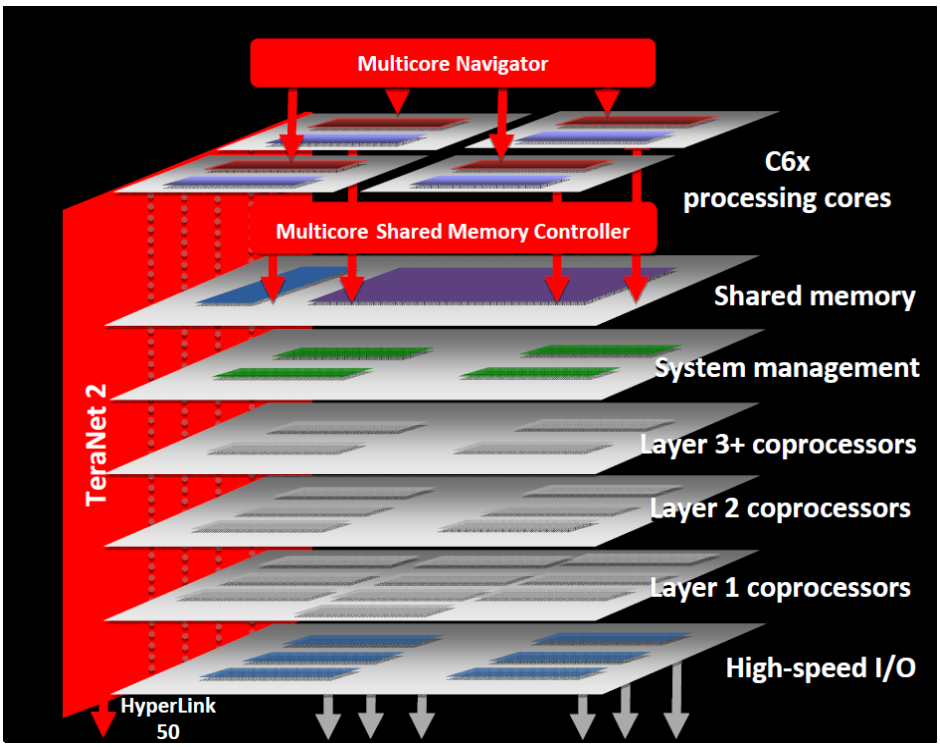


Figure 2.2: Exemple d'un MPSoC à plusieurs couches de `TEXAS INSTRUMENT` [119].

À l'opposé des systèmes électroniques à usage général, tels que les ordinateurs, un MPSoC cible toujours une application particulière bien définie. Actuellement, une architecture multicouche pour un MPSoC apparaît comme un nouveau concept pour augmenter le taux d'intégration des technologies microélectroniques. Une architecture multicouche intègre une couche logicielle qui facilite la programmation et le contrôle de systèmes, en offrant de meilleures performances qu'un MPSoC traditionnel. L'entreprise `Texas Instrument` (TI) [120] a présenté sa nouvelle architecture multicouche, illustrée par la Figure 2.2. Elle est composée de plusieurs couches, chacune contenant un type de ressources physiques (processeurs, coprocesseurs, mémoire partagée, entrées/sorties etc.).

Avec l'apparition des architectures multicouches, choisir les bonnes configurations est devenu fastidieux. Cependant, la problématique reste toujours la même :

- exploration rapide de l'espace de conception ;

- sélection des solutions les plus appropriées.

Des travaux récents dédiés à l'analyse et l'exploration, au niveau système, d'architectures multichouches émergent [102, 39, 80]. Par exemple, Doan et al. [39] proposent l'utilisation des méta-heuristiques pour explorer rapidement des espaces immenses de solutions possibles. Ils utilisent une méthode approximative pour déterminer l'ensemble des solutions possibles. Une fois cet ensemble approximé, les auteurs présentent un modèle de préférence pour réduire l'espace de solutions. Ce modèle prend en considération plusieurs critères qui affectent les performances telles que : la longueur des interconnexions, le coût de fabrication, le volume du produit final, la consommation d'énergie et la dissipation de la chaleur.

2.2.2 Caractéristiques d'un MPSoC

Un MPSoC est souvent dédié à l'exécution d'applications bien spécifiques. Il est cependant soumis à des contraintes physiques car les ressources peuvent être plus restreintes que dans un système fixe : par exemple le nombre et la fréquence des processeurs, la taille des mémoires. D'une part, il nécessite des ressources physiques puissantes pour exécuter des applications hautes performances. D'autre part, il est généralement produit en grande quantité et a souvent la batterie comme source d'alimentation primaire. Pour cela, la consommation d'énergie et la dissipation de la chaleur doivent être minimisées autant que possible.

Afin de trouver des configurations optimales d'un MPSoC, une analyse du système à un niveau d'abstraction unique n'est pas appréciée. Il faut combiner dans le processus de conception plusieurs niveaux d'abstraction pour réussir enfin de compte à obtenir une configuration du système qui respecte certains critères. Ces derniers sont toujours les mêmes :

- **réduction de la consommation d'énergie** : un MPSoC doit généralement être optimisé pour la consommation d'énergie. Cela est dû au manque de source d'alimentation. Dans le cas contraire, son autonomie diminue et provoque une insatisfaction auprès de l'utilisateur ;
- **fiabilité du système** : certains MPSoC sont de types critiques. Une panne mineure peut avoir des conséquences dramatiques telles que des morts, des dommages matériels importants, ou des conséquences graves pour l'environnement. Par exemples, la disponibilité des appareils médicaux, des contrôleurs de freins des véhicules, des systèmes de navigation avionique doit être toujours garantie ;
- **amélioration des performances** : les performances de beaucoup de MPSoC disponibles aujourd'hui sont suffisamment élevées. Cependant, il y a encore un certain nombre d'applications nécessitant plus de performances, telles que la traduction automatique de voix, la radio logicielle, la reconnaissance d'images animées, les applications *streaming*. Bref, toutes les applications hautes performances ;
- **réduction du temps de mise à disposition sur le marché** : le temps de mise à disposition sur le marché (*time-to-market*) est un facteur important. Il est un des facteurs qui déterminent le profit d'un MPSoC, une fois déployé sur le marché. Afin d'être réactifs aux publications de nouveaux standards (décodages audio par exemple) et de nouveaux besoins, les concepteurs de MPSoC ont un temps d'échéance à respecter pour concevoir et programmer leur système. L'objectif est donc de diffuser sur le marché, le plus rapidement possible (avant les autres si possible), un produit répondant à une nouvelle norme ;

- **réduction du coût de la conception** : la plus grande menace pour l'avenir du MPSoC est son coût de conception. Mis à part les coûts de fabrication, déployé un MPSoC sur le marché comporte plusieurs efforts tels que le marketing, la vente, l'administration et des frais divers. Cependant, le coût de conception reste énorme. Actuellement, le coût de conception chez `Intel Corporation` est de 37%, `Advanced Micro Devices` 53%, `Texas Instrument` 47%, `STMicro-electronics` 63% et `NVIDIA corporation` 54 % [104].

Pour répondre aux critères ci-dessus, il est d'abord nécessaire de mettre en place des méthodes et des outils facilitant la conception de MPSoC et réduisant le coût de conception. En plus, il est souhaitable de réaliser la conception à un niveau d'abstraction élevé. Cela permet, dans un premier lieu, de réduire la quantité de détails que le concepteur aura à manipuler.

2.2.3 Intérêt de MPSoC pour le traitement de données intensif dans le cadre de cette thèse

Les applications de traitement intensif de données deviennent de plus en plus répandues. Elles mettent en jeu un volume considérable de données. Elles exigent une grande puissance de calcul afin de satisfaire des contraintes de vitesse de traitements et de performances. Les architectures parallèles semblent être la voie la plus prometteuse pour répondre à la complexité croissante de telles applications. Les MPSoC s'avèrent alors convenables pour ce genre de traitement.

Nous proposons, dans le cadre de cette thèse, l'utilisation du profil UML/ Marte pour la modélisation d'applications hautes performances, d'architectures massivement parallèles, et de l'associations des deux dernières.

Afin d'analyser et de vérifier les systèmes résultants d'une modélisation Marte, nous définissons une abstraction de modèles Marte, par le biais d'horloges abstraites inspirées de ceux des langages réactifs synchrones. Les traces d'horloges abstraites capturent les comportements de systèmes en représentant l'activité des unités de traitements durant l'exécution de fonctionnalités. Une technique d'analyse, également inspirée de l'approche synchrone, est définie. Cette technique permet de vérifier des contraintes fonctionnelles telles que les dépendances de données induites par un modèle Marte reflétant un ordre d'exécution de tâches. En outre, elle permet d'analyser des contraintes non fonctionnelles telles que l'estimation de temps d'exécution, le respect des temps d'échéance et l'estimation de la consommation d'énergie. L'intérêt de notre approche est de réduire, rapidement et très tôt dans le flot de conception, l'ensemble des configurations possibles d'architectures et d'associations.

2.3 Méthodologies de conception

Durant la phase de conception d'un MPSoC, le temps et la précision des simulations dépendent énormément du niveau d'abstraction considéré : la simulation au niveau transactionnel (TLM) est d'environ 400 fois plus rapide (voir plus selon la taille de l'application) que la simulation au niveau transfert de registre (RTL) [21]. Cela est dû à la grande quantité de détails architecturaux qui est prise en compte au niveau RTL par rapport au niveau TLM. Par exemple, l'implémentation d'un algorithme sur une architecture à partir de mauvais choix de configurations a des conséquences graves sur le processus de développement. Il est très probable de recommencer à nouveau le

processus de développement avec une nouvelle configuration. Cela augmente le temps de mise à disposition du MPSoC sur le marché et impose des coûts financiers énormes.

Par conséquent, un compromis existe entre, d'une part, la précision de l'analyse, de la vérification et de la simulation du système, et d'une autre part, le temps global de la mise à disposition sur le marché.

2.3.1 Niveaux d'abstraction : le modèle Y proposé par Gajski et Kuhn

Au cours des dix dernières années, beaucoup d'environnements de conception de MPSoC, au niveau système, ont été développés [107]. Leur but est de faciliter la conception de systèmes complexes et d'éliminer très tôt, dans le flot de conception, de mauvaises configurations.

Des modèles, conçus au niveau système, sont raffinés dans une étape suivante vers le niveau RTL. Ce raffinement, d'un niveau de spécification abstrait vers des niveaux plus détaillés, peut concerner les données manipulées ou les communications. Plusieurs travaux, existant dans la littérature, facilitent le passage d'une description algorithmique ou transactionnelle vers une description RTL plus détaillée [124, 34, 19].

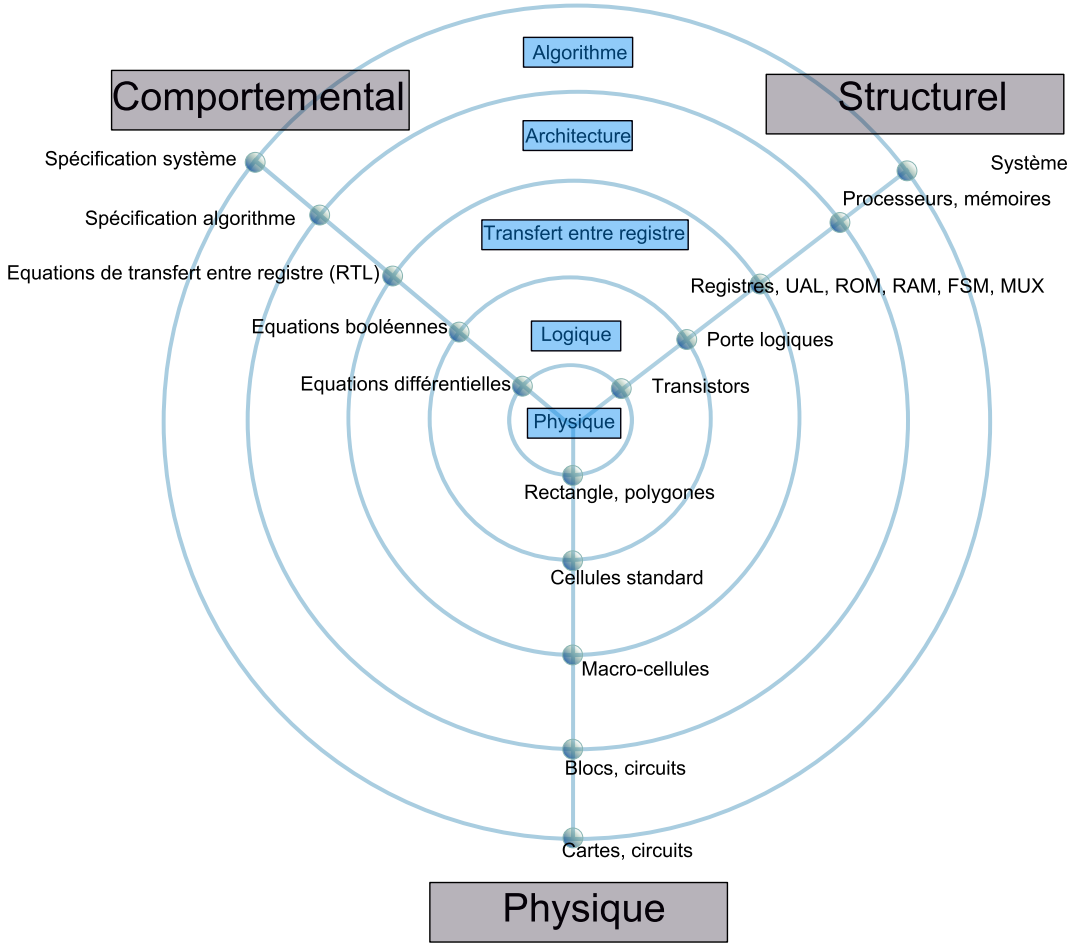


Figure 2.3: Trois types de modèles et leurs degrés de finesse selon un diagramme dit *en Y*.

2.3.1.1 Vision globale

La méthodologie de conception selon les différents niveaux d'abstractions peut être observée dans le modèle Y, représenté par la Figure 2.3, proposé par Gajski et Kuhn en 1983 [48]. Sur chaque axe sont reportés les différents niveaux du domaine de description. Ils sont d'autant plus abstraits que l'on s'éloigne du centre du Y. Les cercles représentent les niveaux d'abstractions. Les axes représentent les trois domaines (ou vues) selon lesquels on peut modéliser un système électronique. Chacun de ces domaines possède plusieurs niveaux d'abstraction selon la finesse des détails présentés :

- **comportemental** : il ne décrit que les fonctionnalités d'un système et non pas la manière dont il est conçu. C'est le domaine de description le plus abstrait ;
- **structurel** : il représente un système sous forme de sous systèmes interconnectés. Le système est vu comme étant un assemblage de composants. On décrit les interconnexions entre les différents sous-ensembles, qui peuvent eux-mêmes être composés d'autres sous-ensembles jusqu'à arriver au plus bas niveau qui est le transistor;
- **physique** : il décrit physiquement le système. Le concepteur connaît par exemple les dimensions et le type des matériaux, les transistors, les masques (polysilicium, dopant N, métal, etc.).

Le modèle Y, adopté par de nombreux travaux récents [7, 97, 59, 95], propose une conception conjointe de MPSoC en faisant une séparation claire entre une application, une architecture, et l'association entre une application et une architecture. Dans cette vision, une application décrivant des fonctionnalités est complètement indépendante de l'architecture qui va l'implémenter. C'est durant l'association d'une application et d'une architecture que les deux vues séparées s'entrelacent.

Cette forte séparation entre l'application et l'architecture peut être aussi vue comme une séparation entre la quantité de travail (*workload*) dans la partie application et les ressources physiques. En effet, le traitement d'une application impose une charge de travail sur les ressources physiques définies dans une architecture. Cette charge de travail se divise en deux parties : une partie représentant le traitement de données (nécessitant des unités de calculs) et une autre partie représentant des communications (nécessitant des unités de communications et de stockages). Les ressources physiques peuvent donc être dédiées au calcul, à la communication ou au stockage.

2.3.1.2 Niveaux d'abstraction

Durant la conception de MPSoC, une description à un haut niveau d'abstraction (contenant des informations fonctionnelles explicites) ne contient pas autant de détails d'implémentation qu'une description de bas niveau [25, 28]. La Figure 2.4 montre les différents niveaux d'abstraction du haut vers le bas. Au niveau le plus abstrait se trouve le concepteur alors que l'électronicien qui fabrique la puce est au plus bas niveau.

La conception de MPSoC peut être décrite en utilisant différents langages de programmation et selon différents niveaux : *algorithmique, architectural, transfert de registre, logique* ou *physique*. Dans le but de concevoir efficacement des systèmes sur puces, il faut connaître les possibilités réelles d'optimisation dans tous les niveaux d'abstraction. Cette optimisation concerne des choix de conception qui affectent les performances d'un système telles que l'énergie dissipée, le temps d'exécution etc.

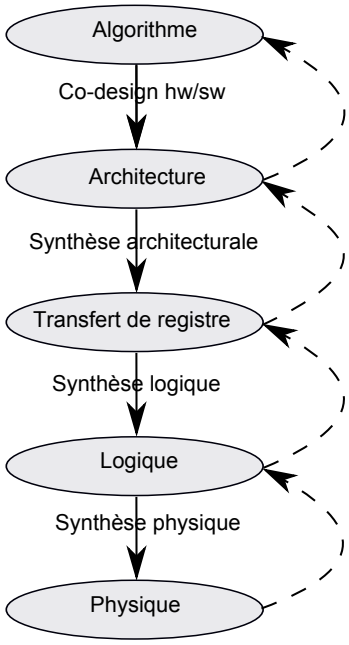


Figure 2.4: Niveaux d'abstraction selon Gajski et Kuhn.

Niveau algorithme

Le niveau algorithme représente la phase initiale du développement d'un système. Il prend en compte des exigences fonctionnelles et des contraintes de réalisations de systèmes. La plupart des algorithmes sont souvent déjà disponibles, soit parce qu'ils ont été déjà écrits dans un autre contexte, soit parce qu'ils font l'objet d'une norme (comme pour les algorithmes de codage/décodage vidéo). À ce niveau-là, le logiciel et le matériel ne sont pas encore distingués. Les systèmes sont développés à l'aide de langages de haut niveau, comme UML [85], Matlab [76] ou C++ [114].

Niveau transactionnel (architectural)

Au niveau transactionnel, la description d'architecture est ajoutée. À ce niveau d'abstraction, la façon dont sera réalisé chaque module n'est pas encore fixée. Par exemple, tous les transferts de données peuvent se faire par un même canal de communication. On distingue deux types de modèle : les modèles transactionnels purement fonctionnels, parfois nommés PV (*Programmer View*) et les modèles transactionnels temporisés, parfois nommés PVT (*Programmer View Timed*).

Le modèle transactionnel non temporisé cible la simulation de système et la vérification de propriétés fonctionnelles très tôt dans le flot de conception. L'objectif de ce modèle est d'obtenir des simulations rapides. En revanche, le modèle transactionnel temporisé contient des annotations temporelles essentielles pour caractériser la communication et le comportement de systèmes. L'objectif de ce modèle est d'ajouter plus de précision dans les simulations afin d'analyser l'architecture considérée. Pour écrire un modèle transactionnel, il est nécessaire d'avoir déjà une idée du partitionnement logiciel/matériel. Les modèles transactionnels temporisés permettent par exemple d'évaluer de façon précoce les choix d'architecture et de partitionnement, et donc de revoir certains choix lorsque cela apparaît nécessaire.

Niveau transfert de registre

Le niveau transfert de registre (RTL) reflète la modélisation du circuit. Il comporte une description précise, au bit et au cycle près, des transferts de données synchrones et des différentes unités fonctionnelles, comme par exemple les multiplicateurs, les unités arithmétiques et logiques ainsi qu'une description du contenu des registres. Ce niveau de modélisation est largement utilisé dans des langages d'architecture tels que Verilog [122] et VHDL [61].

Ce qui distingue le niveau RTL des autres niveaux plus hauts est la précision aux niveaux cycles d'horloges du processeur ainsi que la description détaillée des transferts de registre et des logiques combinatoires. Il est en effet possible de déterminer, à partir d'une description RTL, les cycles d'horloges processeur au cours de laquelle chaque opération a lieu. En effet, les modules RTL sont très précis permettant directement une mise en œuvre réelle sur du matériel re-configurable tel que les FPGAs [94].

Niveau portes logiques

Ce niveau décrit le système au niveau de portes logiques construites à partir de plusieurs transistors reliés entre eux. Il est utilisé par des outils de synthèse pour optimiser le matériel. Normalement, les concepteurs de matériels ne travaillent pas à ce niveau. Des outils commerciaux bien connus dans l'industrie permettent de produire automatiquement de telles descriptions structurelles. Ce passage est appelé synthèse RTL.

Niveau physique

Ce niveau décrit le fonctionnement ou la structure d'un système au niveau des éléments électriques (transistors, résistances, capacités, sources, etc.). Le niveau logique décrit précédemment est transformé vers une description au niveau physique en vue d'avoir le placement physique des transistors et leurs interconnexions sur le dispositif physique avant la phase de fabrication. Cette description physique de bas niveau est généralement utilisée pour vérifier si la conception finale du système est cohérente avec les spécifications de temps et de fréquence.

2.3.2 Flots de conception

Face à un marché en perpétuelle progression, les fabricants de MPSoC doivent faire preuve d'une grande réactivité. Ils doivent ainsi continuellement améliorer leurs techniques de conception et de fabrication de MPSoC afin de les diffuser sur le marché le plus rapidement possible ; la contrainte de la mise à disposition sur le marché étant au cœur de tout nouveau projet.

Dans le but de faire face à la complexité croissante de la conception de MPSoC, la notion d'abstraction s'avère la solution la plus efficace. Au niveau système par exemple, le concepteur peut traiter des applications hautes performances avec une maîtrise totale des technologies. Il n'a plus à se soucier de la description explicite du niveau physique. Cependant, des questions se posent sur le style de développement :

- est-ce que nous partons du plus haut niveau et descendons vers le bas?
- procédons-nous à l'envers en commençant la conception au plus bas niveau et en remontant dans les niveaux d'abstraction?

Ces deux questions méthodologiques ont fait l'objet de plusieurs travaux de recherches. Elles sont connues sous le nom de *top-down* et *bottom-up*.

Dans de nombreux cas, l'approche *top-down* reflète le synonyme de l'analyse et de la décomposition alors que l'approche *bottom-up* reflète l'abstraction. Dans d'autres cas, la conception des systèmes peut être une combinaison des deux approches. C'est ce qu'on appelle *meet-in-the-middle*. Un rapport de recherche, fait par le département informatique de l'université de Namur [99], résume des informations fournies par 54 doctorants en 2009. Ces doctorants sont tous spécialistes dans le domaine de conceptions de systèmes embarqués. En ce qui concerne la stratégie de conception de systèmes :

- 56.5 % des doctorants ont choisi l'approche *top-down* ;
- 19.6 % ont choisi l'approche *meet-in-the-middle* ;
- 15.2 % ont choisi l'approche *bottom-up*.

Approche de conception descendante (*top-down*)

L'approche descendante offre une planification et une conceptualisation complète et rapide de systèmes [81, 33, 67]. La conception commence par une description très abstraite du système pour être enrichie ensuite par des détails d'implémentations. L'idée de cette approche est qu'aucun code ne peut commencer à apparaître avant qu'un niveau de détail suffisant soit atteint durant le flot de conception. La méthodologie consiste à décomposer le système en des sous-systèmes aisément manipulables et compréhensibles. Elle permet donc d'avoir une vue globale du projet final et de donner une estimation rapide, bien qu'approximative, de la complexité d'un système et de son coût. Cependant, cette méthodologie retarde le test du fonctionnement global de systèmes jusqu'à ce que les détails nécessaires soient présents. Le concepteur est obligé de prendre des décisions de conception prématurées et rend beaucoup plus difficile la correction du design. Traditionnellement, les processus de conception de système à haut niveau ont été présentés selon le modèle Vee [23], qui est centré sur l'approche *top-down*. La faiblesse majeure de cette approche vient essentiellement du fait qu'il existe une grande différence entre le plus haut niveau (fonctionnel) et le plus bas niveau (circuit).

Approche de conception ascendante (*bottom-up*)

L'approche *bottom-up* de MPSoC est une méthodologie traditionnelle de conception [68, 26]. C'est une méthodologie ascendante dans laquelle le concepteur s'appuie sur des composants élémentaires sous forme de blocs qu'on associe à leur tour les uns aux autres jusqu'à parvenir à un système complet. Chaque composant est conçu et vérifié individuellement, très tôt dans le flot de conception [68]. L'idée sous-jacente est de tester le système en commençant par les caractéristiques les plus simples puis de tester à la fin le système en entier.

L'inconvénient majeur de cette méthodologie est la difficulté de simuler l'ensemble d'un système. En effet, plus le système comporte un nombre élevé de composants, plus les temps de simulation sont longs et coûteux en terme de capacité de calculs. Cela rend bien évidemment difficile la vérification du système dans sa totalité. Généralement, chaque bloc est vérifié individuellement en fonction des spécifications qui lui sont propres. Toute erreur constatée oblige le concepteur à reprendre son travail depuis le début. Cela est d'autant plus pénible et coûteux que l'erreur est détectée tardivement. Pour vérifier le fonctionnement du système dans sa globalité, les concepteurs ont le plus

souvent recours à la mesure directe sur des prototypes. Cependant, la réutilisation de l'information est l'un des principaux avantages de cette approche.

Approche de conception *meet-in-the-middle*

Le terme *meet-in-the-middle* a été utilisé la première fois par `Hugo De Man` dans le début des années 1980 [113]. L'approche de conception *meet-in-the-middle* est une mise en pratique simultanée des deux approches de conception *top-down* et *bottom-up*. En effet, il se peut que le concepteur souhaite observer plusieurs comportements d'un système ou sous-système à des niveaux de précisions différents. Le choix du niveau de précision requis pour ces comportements représente d'ailleurs la problématique principale du plan de modélisation. En général, une méthodologie *meet-in-the-middle* applique une méthodologie *top-down* pour des conceptions de niveaux d'abstraction élevés et une méthodologie *bottom-up* pour des conceptions de bas niveaux d'abstraction [35].

Son principal inconvénient est le risque de ne pas pouvoir profiter de la jointure des deux approches au bon moment. Cela est dû à la grande différence des niveaux de détails entre les deux approches ascendante et descendante.

La conception à base de plateforme (ou en anglais *Platform Based Design*–PBD) est actuellement considérée comme étant une approche de conception *meet-in-the-middle*. Durant le processus de conception, des raffinements successifs de la spécification s'entrelacent avec des abstractions d'implémentations possibles [103].

Flot de conception	Description	Avantages	Inconvénients
top-down	raffinements successifs du système étape par étape	planification et compréhension complète du système	décisions de conception prématurées
bottom-up	assemblage de composants élémentaires	analyse et vérification très tôt dans le flot de conception	difficulté de simuler l'ensemble d'un système
meet-in-the-middle	mise en pratique simultanée des deux approches de conception *top-down* et *bottom-up*	adaptée pour la conception des systèmes embarqués complexes et hétérogènes	difficulté de joindre les deux approches *top-down* et *bottom-up*

Table 2.1: Avantages et inconvénients des approches *top-down*, *bottom-up* et *meet-in-the-middle*.

Le Tableau 2.1 présente un résumé des avantages et des inconvénients des approches de conception *top-down*, *bottom-up* et *meet-in-the-middle*.

2.3.3 Revue de travaux existants

Les considérations exposées dans la section 2.3 démontrent l'importance de la simulation complète d'un MPSoC au niveau système ainsi que la mise en œuvre du flot de conception *top-down*. L'analyse et la simulation de MPSoC à ce niveau d'abstraction sont récemment rendues possibles grâce aux langages de modélisation et aux environnements d'analyse et de simulation.

Dans cette section, nous présentons un état de l'art de certains travaux effectués dans le domaine de conceptions et d'analyses de MPSoC. Ces travaux seront critiqués objectivement, ce qui va permettre au lecteur de percevoir la différence entre, d'une part, ces méthodes de conceptions et d'analyses et, d'autre part, la méthodologie de conception et d'analyse de MPSoC que nous proposons dans cette thèse.

Nous présentons d'abord le langage de modélisation UML (*Unified Modeling Language*) qui est très répandu dans la modélisation de systèmes ainsi que des extensions sous forme de profils. Ensuite, nous présentons des modèles de calcul dédiés aux traitements de signal intensif. Nous décrivons par la suite des environnements de conception à base de modèles et de plateforme qui utilisent certains de ces modèles de calcul. Enfin, des outils pour l'exploration de l'espace de conception sont présentés.

UML (*Unified Modeling Language*)

Standardisé par l'OMG (Object Management Group), UML [85] est dédié à la modélisation de n'importe quel système informatique. Sa force est de permettre de schématiser pratiquement tout et que ses modèles sont compréhensibles par une grande partie de gens. Son principal inconvénient est qu'il est très général. En particulier, il existe un grand nombre de points de variation sémantique, où la signification est laissée à l'appréciation du concepteur. D'où l'introduction de la notion de profil UML qui a apparue dans la version UML 1.3. Un profil est un moyen permettant de raffiner la sémantique de UML pour la modélisation de systèmes bien précis :

- le profil `SPT` (*Schedulability, Performance, and Time*) [77] est défini comme support à UML dans le but de modéliser certains domaines temps réel. SPT introduit une notion quantifiable du temps et de ressources. Il décrit ainsi les éléments d'un système avec des concepts quantitatifs de temps, de performance et d'ordonnançabilité permettant l'analyse de systèmes temps réel. Cependant, SPT ne considère qu'une échelle de temps chronométrique ne faisant référence qu'au temps physique ;

- le profil `Marte` (*Modeling and Analysis of Real-time and Embedded systems*) [89] est dédié à la modélisation des systèmes embarqués temps réel. Il offre la possibilité de modéliser des applications, des architectures et les relations entre elles. Il offre aussi des extensions pour faire de l'analyse de performances et d'ordonnancements en introduisant une notion quantifiable de temps physique et logique (ce profil est largement détaillé dans le chapitre 4) ;

- le profil `SysML` (*System Modeling Language*) [86] étend UML par de nouveaux concepts et nouveaux types de diagrammes. Les diagrammes de séquence et les machines à états sont adoptés par SysML par exemple. En plus de ces derniers, nous retrouvons dans SysML un diagramme paramétrique et un autre décrivant les exigences d'un système ;

- le profil `UML4SystemC/C` [96] propose un ensemble de concepts proches de SystemC. Il est dédié à la co-modélisation de MPsoC et facilite le prototypage au niveau transfert de registre. Le passage d'une modélisation de haut niveau d'un système vers son implémentation et prototypage est envisageable par le biais de transformations de modèles. Le profil UML4SoC [84] vise aussi le même objectif.

Bien que la plupart des profils ci-dessus proposent des concepts utiles pour la conception de MPSoC, ils ne prennent pas suffisamment en considération certains aspects cruciaux

tels que l'analyse temporelle et la possibilité de spécifier du parallélisme massif. À notre connaissance, Marte est le seul profil qui offre un ensemble riche et satisfaisant de concepts pour la conception de MPSoC à haut niveau.

Nom	Cible	Modélisation
SPT	système temps réel	temps, ordonnancement, performance
Marte	système temps réel	fonctionnalité, architecture, association, temps, ordonnancement, performance
SysML	système général	spécification, analyse, vérification
UML4SoC	système sur puce	concepts proches de SystemC

Table 2.2: Classification de profils UML.

Modèles de calcul

Un modèle de calcul (MoC ou *Model of Computation*) représente un ensemble de lois qui régissent les interactions de composants dans un modèle. Plus généralement, c'est un ensemble de règles abstraites permettant au concepteur de créer des modèles selon une sémantique bien précise. Les modèles de calcul les plus intéressants pour modéliser des systèmes de hautes performances (ou de traitement intensif de signal) doivent être en mesure de gérer la concurrence de composants, la réactivité et le temps d'exécution.

Classification des modèles de calcul

Afin de classifier les modèles de calcul, nous nous basons sur plusieurs critères et caractéristiques pour faire la comparaison :

- **méthode de saisie d'information :**
  - `graphique` : les éléments de programme sont définis et manipulés graphiquement plutôt qu'en les indiquant textuellement. La plupart des langages de programmation graphique sont basés sur l'idée des boîtes, des flèches, lignes ou arcs ;
  - `textuel` : le modèle ou programme est codé textuellement ;
- **style de programmation :**
  - `fonctionnel` : un langage fonctionnel a l'avantage d'exprimer directement des fonctions d'accès par des expressions mathématiques. Il peut être défini sous une forme textuelle, graphique ou un mélange des deux derniers ;
  - `flot de données` : les données sont des entités actives qui traversent le programme de manière asynchrone. Les programmes flots de données sont généralement représentés visuellement sous forme de graphes. Un nœud représente une opération. Les données, représentant des entrées aux nœuds, circulent sur des arcs. Cette approche est largement utilisée dans la description des applications de traitement de signal ;
  - `impératif` : les opérations sont décrites en termes de séquences d'instructions qui modifient l'état du programme. Pour ce type de langage, une représentation textuelle est normalement plus appropriée ;

Modèles de calcul fonctionnel

`Alpha.` Le langage Alpha [132] permet d'écrire des systèmes d'équations récurrentes affines. C'est un langage déclaratif, autorisant les équations conditionnelles, où les domaines de définition sont des unions de polytopes paramétrés. L'intérêt d'Alpha est qu'il fournit un langage de haut niveau pour synthétiser des architectures VLSI (abréviation de *Very Large Scale Integration*). Il est basé sur le modèle polyédral, à assignation unique, et est fortement typé. Cela permet de représenter des algorithmes sous forme d'équations récurrentes.

`Sisal.` Sisal (abréviation de *Streams and Iteration in a Single Assignment Language*) [52] est un langage textuel à assignation unique conçu dans l'objectif de fournir une interface utilisateur d'usage générale pour une large plage des plateformes parallèles. Ces sémantiques précisent la dynamique d'un graphe de flot de données et les expressions Sisal évaluent et renvoient des valeurs basées uniquement sur les valeurs limitées à leurs arguments formels et identificateurs constitutifs. La définition initiale de Sisal est basée sur un modèle utilisant des tableaux mono-dimensionnels. Dans cette définition, les tableaux peuvent être construits par la concaténation des tableaux des dimensions intérieures pour construire un seul tableau mono-dimensionnel, qui favorise certaines optimisations comme la vectorisation. Les désavantages de cette approche sont le fait que les tableaux doivent être gardés dans une zone de mémoire continue et que les vecteurs inférieurs doivent avoir la même taille.

Modèles de calcul flot de données

`KPN (Kahn Process Networks).` Les réseaux de processus de Kahn (KPN, ou réseaux de processus) [64] représentent un modèle de calcul déterministe. Les communications se font exclusivement à travers des files d'attentes unidirectionnelles et non bornées. Les deux seules primitives de communication à la disposition des processus sont l'écriture et la lecture (bloquante quand la file est vide) dans une file d'attente. Un processus peut ainsi être vu comme une application de ses flux d'entrée vers ses flux de sortie. Ce modèle est assez souple mais il a l'inconvénient d'être difficile à ordonnancer. Le modèle a été à l'origine développé pour modéliser des systèmes répartis mais a prouvé sa convenance aussi pour les systèmes temps réels en général.

`SDF.` Le modèle SDF (abréviation de *Synchronous Data Flow*) a été créé et développé par Lee et Messerschmitt [72]. En SDF, une application est décrite par un graphe orienté dont chaque nœud consomme et produit des données respectivement sur ses arêtes entrantes et sortantes. Le nombre de données consommées et produites par un nœud à chaque exécution est fixé dès la conception de l'application. Ce nombre doit être connu lors de la modélisation. L'ordonnancement des exécutions, séquentielles ou parallèles, est calculé statiquement. Des applications décrites en SDF ont l'avantage majeur qu'elles sont définies et ordonnancées statiquement. Cependant, la mono-dimenisonnalité du modèle réduit énormément l'ensemble d'applications modélisables. D'autres langages développés à partir de SDF ont essayé d'élargir son expressivité, mais en restant compatible avec SDF.

`FSM.` Les machines à états finis FSM [53] (abréviation de *Finate State Machine*) sont des graphes explicitant des états. Chaque sommet dans un graphe est un état et chaque arc, appelé transition, définit une condition pour passer d'un état à un autre. Les traitements,

MoC	Structure de donnée	Accès donnée	style	Saisie information
SDF	multidimensionnel	tableau	flot de donnée	graphique
Array-OL	multidimensionnel	tableau	flot de donnée	graphique
Stream-IT	monodimensionnel	tableau	impérative	textuel
ALPHA	polyhedron	affine	fonctionnel	textuel
Sisal	multidimensionnel	tableau	fonctionnel	textuel
Sac	multidimensionnel	tableau	impératif	textuel

Table 2.3: Classification des modèles de calcul.

de types atomiques, sont soit liés aux transitions soit aux états. Les arcs définissent donc une séquence entre les traitements. Ces derniers, de type atomique, peuvent effectuer des lectures et des écritures dans des variables partagées par l'ensemble du graphe. L'intérêt des machines à états finis consiste justement dans le fait qu'elles font apparaître un nombre fini d'états. Il est ainsi possible de vérifier certaines propriétés fonctionnelles telles que le passage d'un état à un autre.

Array-OL. Array-OL [22, 38] est un langage pour la description de systèmes d'équations récurrentes uniformes. Il se base sur une représentation graphique ; en cela, il rejoint des modèles tels que les graphes SDF multi-dimensionnels. Les flots de données sont représentés par des tableaux multi-dimensionnels, toriques et non-paramétrés, et éventuellement infinis sur une dimension au plus. Si le temps n'est pas spécifié à priori, comme dans Alpha, il est généralement implicite : la dimension infinie sert souvent à représenter, dans la pratique, une séquence de tableaux finis sur les autres dimensions.

Modèles de calcul impératif

StreamIt. StreamIt [121] est une exception dans les langages textuels à flots de données. StreamIt, pour *stream-MIT*, est un langage impératif utilisant la programmation orientée objet, développé au *Massachusetts Institute of Technology* spécialement pour des systèmes modernes orientés flots de données. Le langage est conçu pour faciliter la programmation des applications complexes de flot de données et l'infrastructure de compilation disponible pour le langage vise un placement efficace sur des architectures matérielles.

SaC. SaC [105] (abréviation de *Single Assignment C*) est un langage de programmation de tableaux. Il est dédié aux domaines de traitement intensif de données tels que le traitement de signal. Sa particularité est qu'il combine les avantages d'un programme de haut niveau et d'une efficacité d'exécution du modèle fonctionnel sous-jacent qui est la brique de base de l'expression du parallélisme. Il partage des principes de conception avec Sisal, mais introduit des tableaux n-dimensionnels comme structures de données principales.

Conception à base de modèles

La conception de MPSoC à base de modèles devient de plus en plus répandue vu que la notion de modèle simplifie la description du comportement du système en cachant les détails d'implémentation de bas niveau. Ces détails ne sont pas essentiels dans les premières phases de conception. Dans ce genre de programmation, le modèle de calcul décrivant les fonctionnalités des applications est d'une grande importance. Nous

présentons, par la suite, un état de l'art sur des environnements de conception de MPSoC à base de modèles :

- `Metropolis` [10] propose un méta-modèle qui supporte plusieurs types de modèles de calculs actuels et pourra en accueillir d'autres. Il permet de spécifier des applications et des architectures séparément ainsi que l'association entre ces deux derniers. Aussi, des techniques de vérification formelle et de simulation de modèles sont définies ;

- `Ptolemy` [70] est un environnement de simulations et de prototypages de systèmes hétérogènes. Il supporte lui aussi plusieurs modèles de calcul différents. En utilisant des techniques telles que le raffinement et la composition hiérarchique, le concepteur peut spécifier et simuler des applications ayant différents modèles de calcul et des architectures hétérogènes à différents niveaux d'abstractions. Parmi les différents modèles de calculs soutenus, nous citons SDF [71], le modèle réactif synchrone [93], KPN [64] et FSM [53] ;

- `Koski` [65] est lui aussi basé sur le modèle de calcul KPN en partant d'une spécification UML. Koski se sert de plusieurs concepts UML tels que les diagrammes d'état, de classe, de structure et de séquence. Le flot de conception permet une génération automatique de code pour analyser et simuler le système à plusieurs niveaux d'abstraction;

- `Simulink` [110] est dédié à la conception de systèmes embarqués. Il permet la modélisation à base de modèle, la simulation et l'implémentation des systèmes. Son intégration avec l'environnement MATLAB permet d'analyser et de visualiser les résultats et de tester les données. Il est considéré comme un environnement de modélisation et de simulation pour les systèmes embarqués dans plusieurs domaines industriels et notamment celui de l'automobile, de la communication, du contrôle, de traitement de signal et du traitement d'images ;

- `CoFluent Studio` [115] offre une description graphique de l'application et de l'architecture. Cette description ressemble beaucoup à celle de UML. Un outil interne permet la génération automatique de code SystemC à partir de la description textuelle afin d'analyser le comportement. La méthodologie de conception repose aussi sur le modèle Y ou l'application et l'architecture sont modélisées séparément mais peuvent être évaluées ensemble après association pour estimer les performances.

Un des points forts de la conception à base de modèle est que le modèle d'application est généralement indépendant du modèle d'architecture. En conséquence, un modèle d'application peut être simulé sur plusieurs modèles d'architecture facilitant ainsi le travail des concepteurs et favorisant la réutilisation de données.

Platform Based Design (PBD)

Afin de réduire les efforts de conception, la notion de réutilisation d'information est favorisée. La conception à base de plateforme offre cette possibilité. Une architecture donnée peut être utilisée pour l'exécution de plusieurs types d'applications. Cela réduit beaucoup le temps et l'effort en comparaison avec une nouvelle implémentation. Plusieurs environnements se basent sur la conception PBD, nous en citons deux :

- `Artemis` [92] est dédié à la modélisation et la simulation de MPSoC au niveau système. Le flot de conception suit le modèle Y. Le modèle d'application est basé sur les réseaux KPN [64]. Le modèle d'architecture est représenté par un ensemble de composants tels que des processeurs, co-processeurs, mémoires, bus, etc. Après la phase d'association de l'application sur l'architecture, un code VHDL est automatiquement généré qui sera implémenté ultérieurement sur des FPGAs. Le point fort de cette méthode est que le modèle d'application est indépendant du modèle d'architecture et de la phase d'association. En conséquence, un modèle d'application peut être simulé sur plusieurs modèles d'architecture et d'association selon différents niveaux d'abstraction ;

- `Mescal` [75] a introduit une méthodologie pour la production des plateformes d'architecture réutilisables. Ces plateformes sont facilement programmables pour exécuter différents types d'application. En particulier, Mescal aborde le problème de l'exploration de l'espace de conception des applications de traitement réseaux sur des processeurs NISP (abréviation de *Network Instruction Set Processor*). La partie application décrivant les fonctionnalités est représentée par Ptolemy alors que la description de l'architecture est faite à différents niveaux d'abstractions. L'association des fonctionnalités sur l'architecture consiste à trouver la bonne combinaison entre les composants fonctionnels et les processeurs d'une part et les communications inter-processus avec les ressources de communications disponibles dans la plateforme.

Outils pour l'exploration de l'espace de conception

Les outils pour l'exploration de la conception de MPSoC constituent un domaine de recherche multiforme dont le but global est de concevoir rapidement des systèmes qui soient fiables, performants et efficaces d'un point de vue énergétique et performance :

- `NESSIE` [98] est un outil d'estimation de performances de MPSoC, à un très haut niveau d'abstraction. Il permet d'explorer plusieurs configurations possibles d'applications et d'architectures, automatiquement et à un faible coût. Cela permet de réduire, très tôt dans le flot de conception, l'espace de solutions possibles.

  Les applications sont modélisées en utilisant un modèle de calcul à base des réseaux de Pétri. Cependant, l'environnement permet d'intégrer différents autres types de modèles de calculs. Le modèle de calcul a été étendu par quelques concepts clés afin de rendre la modélisation de fonctionnalités plus réaliste et pour garantir un comportement déterministe (par exemple, fixation de la taille des données consommées et produites par entités fonctionnelles, spécification des entités fonctionnelles qui déclenchent du travail).

  La modélisation d'architectures dans NESSIE comprend les concepts suivants :

  - des processeurs (ou cœurs) ayant toujours un des cinq états suivants :
    1. *idle* : le processeur est libre et prêt ;
    2. *dormant* : le processeur est éteint ;
    3. *transmission* : transfert de donnée d'un processeur à un autre ;
    4. *mémorisation* : en cours de sauvegarde de donnée en mémoire ;
    5. *traitement* : en cours d'exécution de fonctionnalités.

	Metropolis	Artemis	MMAlpha	SystemCoDesigner	Ptolemy
Application	automobile, communication, traitement d'image	multimédia	Algorithme régulier	automobile, réseau, traitement de signal	traitement de signal
Développeur	Cadence Berkley lab.	Delft Univ. of Technology, Univ. of Amsterdam, Leiden Univ. Philips Eindhoven	Projet COSI, IRISA	Univ. of Erlangen-Nurmberg	Univ. of California Berkeley
Modèle Y	oui	oui	non	non	non
MoC	KPN	KPN/Matlab	Alpha	SystemC	Java, UML
Méthodologie	PBD	MBD	-	-	MBD
Niveau d'abstraction	système	système	système	système	système
Architecture	+	MPSoC	régulière	hétérogène	-

	NESSIE	Mescal	koski	Simulink	CoFluent
Application	-	réseau click	-	contrôle, communication, traitement d'image	-
Développeur	Université Libre de Bruxelles (ULB)	Univ. of california berkley	unknown	Mathworks	Co-Fluent
Modèle Y	oui	oui	oui	non	oui
MoC	Réseau de Pétri	-	KPN/UML	-	Cofluent DSL
Méthodologie	-	PBD	MBD	MBD	MBD
Niveau d'abstraction	système	+	+	système	système
Architecture	MPSoC	-	-	DSP	MPSoC, FPGA

Table 2.4: Classification des environnements de conceptions de MPSoC au niveau système.

un + dans la ligne **Niveau d'abstraction** signifie plusieurs niveaux d'abstraction.
un + dans la ligne **Architecture** signifie plusieurs architectures cibles.
un - dans une case indique une information inconnue.

- des ports associés aux différents processeurs. Ils représentent les interfaces d'entrée et de sortie des différents processeurs. Un port est dans l'un des trois états *envoie*, *réception* ou *inactive* ;
- des connecteurs bidirectionnels ou unidirectionnels permettent de relier différents processeurs communiquant.

Une fois une application et une architecture modélisées par l'utilisateur, l'outil NESSIE offre des techniques d'analyses qui permettent la sélection automatique des processeurs convenables pour l'exécution de fonctionnalités. Cela est fait en définissant un paramètre qu'on appelle *poids d'association*, l'objectif étant de minimiser ce paramètre. Ce dernier peut être une constante ou peut dépendre de plusieurs critères de performances tels que l'énergie, le temps d'exécution, etc.

- `MMAlpha` [36] est basé sur le langage fonctionnel Alpha. Il permet de manipuler des systèmes structurés d'équations récurrentes (uniformes et affines), en s'aidant d'une boîte à outils de transformations de programme (uniformisation, ordonnancement, allocation, etc.). Sous certaines restrictions, l'outil est capable de générer semi-automatiquement, à partir d'une spécification de boucle en Alpha, à la fois une description matérielle (en VHDL) d'un réseau de processeurs (unidimensionnel ou bidimensionnel) et de son interface logicielle/matérielle. L'outil a par ailleurs été validé sur de nombreuses applications, essentiellement issues du traitement du signal.

- `SYSTEMCODESIGNER` [67] a pour objectif d'allouer automatiquement des applications, écrites en SystemC, sur des plateformes MP-SoC hétérogènes. Dans une première étape, le concepteur conçoit un modèle d'application en utilisant SystemC. Dans une deuxième étape, les différentes ressources physiques sont automatiquement générées (sous forme de composants) pour exécuter les fonctionnalités et sont stockées dans une librairie de composants. Une exploration automatique multi-objectifs est appliquée sur le modèle résultant afin d'améliorer les choix de conception. Parmi l'ensemble des solutions optimisées obtenues, le concepteur sélectionne les meilleures configurations qui mènent vers des prototypages rapides. La décision prise pour arriver aux solutions optimales est cependant prise au niveau TLM.

2.4 Analyse de l'état de l'art et positionnement

Les environnements de conception décrits ci-dessous sont bien connus dans le domaine de conceptions de systèmes à haut niveau. La plupart de ces environnements offrent des sémantiques formelles via des méta-modèles, des outils et des méthodologies de conception au niveau système, des techniques de simulations, d'analyses et de synthèses. Les environnements `NESSI`, `Metropolis`, `Koski`, `Artemis`, et `CoFluent Studio` par exemple proposent une séparation complète entre la modélisation des fonctionnalités et de l'architecture suivant en cela le modèle Y. Cependant, `Metropolis` est le seul qui permet la spécification de contraintes de conception au niveau application. Parmi les différentes méthodologies de conception, celles adoptées par `Artemis` et `Koski` se ressemblent beaucoup. En effet, ils spécifient les applications et les architectures dans des modèles séparés. La modélisation de l'application repose sur le modèle de calcul KPN. Ils se servent du modèle KPN afin d'estimer la quantité de travail (*workload*) de l'architecture. On peut aussi remarquer des points de différences. `Artemis` par exemple

utilise Matlab pour décrire l'application et Pearl ou SystemC pour décrire l'architecture alors que `Koski` considère UML comme point d'entrée pour la description de tout le système.

Nous partageons avec ces travaux des similarités et des différences. Le point commun entre la plupart de ces travaux est la séparation de vue. Nous partageons avec `Koski` l'utilisation de UML comme point de départ pour la modélisation du système. Cependant nous utilisons le profil Marte comme support à UML pour décrire des MPSoC massivement parallèles. Nous modélisons à travers UML/ Marte du parallélisme de tâches et de données au niveau application ainsi que des architectures massivement parallèles. Un grand avantage de notre méthodologie de conception est que nous adoptons un langage de modélisation unique, décrivant des applications hautes performances, des architectures massivement parallèles et l'association entre elles.

Nous partageons avec `Artemis` la spécification et l'analyse de contraintes de performance au niveau application. C'est une étape préliminaire à l'analyse du système entier. Il est donc possible d'analyser des exigences de performance d'une application et certaines contraintes de performances en isolant complètement l'architecture.

Une fois le système modélisé en UML/ Marte selon le modèle Y, nous proposons une analyse de son comportement qui, elle aussi, est basée sur le modèle Y. Cette analyse est basée sur la notion d'horloges abstraites. Ces horloges permettent de décrire séparément la quantité de données au niveau application ainsi que leurs dépendances. Une phase d'analyse du comportement fonctionnel est proposée comme dans `Artemis`. Ensuite, une analyse du comportement du système après la phase d'association est proposée. Cette analyse nous permet de vérifier la correction de l'exécution vis-à-vis des contraintes de dépendances de tâches. Elle facilite aussi la vérification de certaines propriétés non fonctionnelles telles que l'estimation des performances et la consommation d'énergie. Cette analyse est offerte par la majorité des environnements décrits ci-dessous. Cependant, l'analyse que nous proposons est faite à un niveau d'abstraction plus élevé, offrant un temps de simulation et un effort presque négligeable en comparaison avec les autres méthodologies d'analyse.

2.5 Discussion

Nous avons vu dans ce chapitre que l'augmentation de la complexité des systèmes électroniques est réelle. Les systèmes peuvent alors supporter de plus en plus de fonctionnalités ce qui rend le processus de conception de plus en plus compliqué. L'une des problématiques liées à cette évolution consiste à s'assurer que le système a été conçu selon les critères souhaités (contraintes fonctionnelles et non fonctionnelles) dans un temps relativement acceptable. Cela impose de mettre en place une méthodologie de conception rigoureuse facilitant l'exploration du domaine de conception.

En premier lieu, on ne peut pas avoir une grande vitesse de simulation et une précision élevée en même temps. En deuxième lieu, les coûts, l'effort et le temps nécessaires pour effectuer la simulation dépendent du niveau d'abstraction de la simulation. Il est donc primordial de développer des méthodologies de conceptions de MPSoC permettant d'évaluer les contraintes et les performances du système dans les premiers stades de flot de conception. Cela réduit l'espace de solutions possible avant de passer aux étapes de simulations de bas niveau (TLM, RTL, etc.).

Chapter 3

Techniques d'analyse de MPSoC

3.1 Introduction

Nous avons présenté dans le chapitre précédent un état de l'art sur le flot de conception de MPSoC en illustré les avantages et les limitations des différents niveaux d'abstraction. Nous avons aussi présenté des outils de vérification et de simulation de MPSoC au niveau système.

Le présent chapitre est dédié à la présentation ainsi qu'à l'étude de différentes techniques d'analyse existantes de MPSoC. Celles qui nous intéressent tout particulièrement sont : l'ordonnancement des tâches, l'analyse de synchronisation et les techniques de diminution de la consommation d'énergie et de gestion mémoire. Nous pensons que ces caractéristiques peuvent largement contribuer à l'amélioration des performances de ces systèmes.

Dans la section 3.2, nous présentons certaines caractéristiques de MPSoC telles que le temps, l'énergie, l'expression du parallélisme et la hiérarchie de mémoire. Dans la section 3.3, nous présentons les techniques et les méthodes d'analyse de MPSoC. Dans la section 3.4, nous détaillons des travaux récents qui analysent des MPSoC et qui explorent leurs espaces de conception selon les caractéristiques définis dans la section 3.2. Enfin, nous concluons dans la section 3.6.

3.2 Modélisation des caractéristiques d'un MPSoC

La modélisation des caractéristiques de MPSoC est une étape primordiale dans le flot de conception conjointe matérielle/logicielle. En effet, un des problèmes majeurs depuis toujours consiste à trouver les meilleurs choix de configurations des ressources physiques. Cela doit être fait tout en respectant des contraintes imposées sur le fonctionnement global du système. Le découpage de fonctionnalités, la configuration de processeurs et de mémoires, l'association et l'ordonnancement de fonctionnalités sur des ressources physiques, la consommation d'énergie globale et le temps d'exécution sont des exemples d'analyse de MPSoC.

3.2.1 Temps

La modélisation du temps dans certains MPSoC, tels que ceux qui fonctionnent en temps réel, est cruciale. Cependant, et contrairement à d'autres domaines de la science et de l'ingénierie, la modélisation du temps dans les systèmes électroniques en générale, et les MPSoC en particulier, est loin d'être une démarche unifiée pour la plupart des analyses systèmes. Cela est dû probablement au fait que la problématique de modélisation du temps se pose dans différents domaines, dans différentes circonstances, et a souvent été traitée d'une manière assez ad hoc.

La notion de temps peut être représentée de différentes manières selon le type de comportement modélisé :

- le temps est pris en compte selon les relations de cause à effet. Le comportement modélisé est dit causal ou fonctionnel ;
- les entrées et les sorties sont produites au même instant, c'est-à-dire que le temps de réaction du système est nul. Le comportement modélisé est dit synchrone ;
- une action spécifique décrit l'unité de temps (fréquemment notée tic). Le comportement modélisé est dit discret ;
- le temps est représenté par des valeurs réelles. Le comportement modélisé est dit continu.

3.2.1.1 Observation d'événements et synchronisation

Durant l'exécution de fonctionnalités d'un système, des événements peuvent être enregistrés.

Definition 3 (Évènement) *Un événement est composé généralement d'un identificateur et d'une estampille temporelle indiquant son instant d'apparition. Un événement est déclenché lors de l'appel à l'exécution de la fonctionnalité qui lui est associée.*

L'analyse d'événements d'un système permet de déduire l'état du système, de vérifier l'ordre d'exécution des événements et de mesurer les performances.

Plusieurs occurrences d'un événement peuvent être enregistrées dans une trace que nous appelons *trace d'événements*. Un système peut donc contenir plusieurs traces d'événements dont chacune est cadencée par une horloge locale à chaque trace. En cas d'absence d'une horloge globale pour l'ensemble des traces, des problèmes de synchronisations entre différents événements peuvent avoir lieu. Il faut donc effectuer une synchronisation des estampilles temporelles des événements. Cette synchronisation

passe d'abord par une reconstruction d'une base de temps globale représentée par une horloge globale (ou horloge de référence).

Durant la conception d'un MPSoC, une des vérifications essentielles est de garantir que l'ordre des événements dans des traces d'événements correspond à l'ordre réel ou l'ordre souhaité. Par exemple, lorsque différentes traces sont analysées conjointement selon une horloge globale, il se peut qu'un événement est déclenché trop tard ou trop tôt par rapport à un autre événement. Ce genre de problème doit être évité afin de respecter l'ordre d'exécution des différents évènements.

3.2.1.2 Horloges physiques et logiques

Durant la conception des MPSoC, notamment ceux qui sont temps réel, il est primordial de prendre en considération l'écoulement du temps, représenté par des horloges. Nous classifions les horloges selon leurs types : physique, logique, vectorielle ou matricielle.

Definition 4 (Horloge physique) *Une horloge physique, représentée généralement dans un système électronique par un cristal de quartz, oscille à une fréquence bien définie indiquant le temps actuel (ou réel).*

Ce type d'horloge permet d'associer, à un événement donné, un temps physique selon une échelle. Cependant, de nombreuses applications nécessitent seulement des informations concernant l'ordre des événements et non pas l'heure exacte de leurs occurrences. Par conséquent, ce type d'horloges n'est pas généralement nécessaire dans certaines analyses (par exemple l'exclusion mutuelle, le débogage et l'optimisation de systèmes distribués, la tolérance aux défaillances) [129].

Dans des systèmes distribués, il est souvent impossible d'avoir des horloges parfaitement synchronisées sur une échelle de temps globale. Il est donc utile de définir un ordre entre les événements du système (représentés sous forme d'horloges). Pour cela, il est possible de définir une relation globale de précédence, l'une des plus répandues étant la relation de causalité définie par Lamport et al. [69]. Les auteurs définissent une relation "*s'est passé avant*" pour capturer des dépendances causales entre différents événements. Il existe divers mécanismes de codage de ces relations d'ordre (horloges logiques de Lamport, horloges vectorielles et matricielles).

Dans le cadre de cette thèse nous nous intéressons à un type bien précis d'horloges logiques : les horloges logiques abstraites [49].

Definition 5 (Horloge logique abstraite) *Une horloge abstraite est caractérisée par deux types d'horloges logiques : une horloge de référence propre au système (la plus rapide en général) et les différentes autres horloges du système dont les instants sont forcément calés sur des instants de l'horloge de référence.*

On repère ainsi l'avancement de l'exécution d'un système avec le temps selon les instants de l'horloge de référence qui lui est associée. Toutes les actions qui sont faites en parallèle partagent une échelle de temps commune, du type discret, qu'on appelle *horloge de référence*. Chaque occurrence d'une action ou d'un événement peut être calée sur cette horloge. Cela supprime la possibilité d'indéterminisme entre des occurrences d'événements.

3.2.2 Énergie

Les MPSoC doivent généralement être conçus de manière à optimiser autant que possible la consommation d'énergie. Cela est dû au manque de capacités énergétiques

d'alimentation. La forte consommation d'énergie diminue l'autonomie du système qui réduit à son tour la durabilité de fonctionnement. Cela aboutit à l'insatisfaction des utilisateurs et, par conséquent provoque une diminution dans la vente.

La consommation totale d'énergie d'un dispositif électronique est en général composée de deux types :

- `statique` : l'énergie est consommée lorsqu'un dispositif est allumé (sous tension) mais il n'y a pas de changement dans les valeurs de signaux ;

- `dynamique` : l'énergie est consommée lorsqu'un dispositif est actif (sous tension et faisant du traitement). Dans ce cas, les valeurs des signaux sont en évolution.

Dans la technologie CMOS (*Complementary Metal Oxide Semiconductor*), la source principale de la consommation d'énergie dynamique est due à la puissance nécessaire pour charger et décharger les capacités de sortie des portes alors que la consommation d'énergie statique est due à la fuite du courant [1].

Afin de pouvoir évaluer les performances d'une architecture donnée, deux critères doivent être considérés : la puissance de calcul nécessaire à l'exécution d'une application et la dissipation d'énergie associée à cette exécution. L'énergie consommée dans un intervalle de temps $[t_1,t_2]$ est par définition l'intégrale de la puissance dissipée sur le temps :

$$E = \int_{t_1}^{t_2} P(t)dt \tag{3.1}$$

où $P(t)dt$ est la puissance dissipée à l'instant infinitésimal dt.

3.2.2.1 Consommation d'énergie statique

La consommation de la puissance statique dans les circuits CMOS a contribué d'une manière significative ces dernières années dans la consommation totale de la puissance [62]. Dans le but de réduire la consommation de la puissance statique résultant de la fuite du courant, une solution primordiale est de faire rentrer le processeur dans le mode dormant (l'éteindre complètement en coupant la tension). Toutefois, la réactivation du processeur impose des coûts du point de vue temps et consommation d'énergie à cause du changement de son état dormant/actif et de la nécessité de récupérer les données des registres ou de la mémoire. Il est donc parfois préférable de perdre de l'énergie statique que d'éteindre et de réactiver continuellement un processeur.

3.2.2.2 Consommation d'énergie dynamique

La puissance dynamique est dissipée lorsque les entrées de données ou d'horloge changent de niveau logique. Elle est due au chargement et au déchargement de données. Dans les circuits CMOS, avec des finesses de gravures supérieures à 0.13 µm, la puissance dynamique représente 80-85 % de la puissance totale dissipée [42]. Cette puissance dynamique est proportionnelle à la fréquence de fonctionnement, à la capacité de la charge et au carré de la tension d'alimentation. Elle est définie par la formule suivante :

$$P_{dynamique} = \alpha \times F \times C \times V^2 \tag{3.2}$$

où α est le taux d'activité c'est à dire le nombre de transitions par cycle d'horloge, F est la fréquence de fonctionnement du processeur, C est la capacité équivalente et V est la tension d'alimentation [27].

On remarque dans l'équation de la puissance dynamique qu'il existe quatre paramètres (α, F, C et V) pour diminuer la puissance dissipée d'un système. Ainsi, toutes les techniques de réduction de la puissance dynamique s'attaquent à l'un ou l'autre de ces paramètres.

En raison de la relation quadratique qui existe entre la tension d'alimentation et la puissance dissipée, la diminution de la tension provoque des gains importants en consommation d'énergie. L'ajustage de la valeur des fréquences/tensions, selon la quantité de travail (*workload*) à exécuter, est actuellement possible dans la plupart des microprocesseurs actuels. En assumant une relation linéaire entre la valeur de la fréquence et de la tension, l'augmentation ou la diminution de la fréquence affecte directement la puissance dynamique dissipée du système.

3.2.3 Niveau de parallélisme

Une machine parallèle est essentiellement un ensemble de processeurs qui coopèrent et communiquent à travers des mémoires partagées ou distribuées. On distingue classiquement quatre types principaux de parallélisme de tâches et/ou de données : SISD, SIMD, MIMD et SPMD. Cette classification permet d'expliquer les bases de fonctionnement des architectures parallèles.

3.2.3.1 Parallélisme de tâches et de données

Le parallélisme de tâches consiste à exécuter plusieurs tâches sur un bloc de données. Chaque tâche effectuée en parallèle est considérée comme une fonctionnalité autonome. Quand au parallélisme de données, une tâche est exécutée sur plusieurs processeurs. Ces deux types de parallélisme apparaissent naturellement dans plusieurs types d'applications : le calcul itératif sur des tableaux, le calcul en flux (*stream computing*) comme la compression, le cryptage, le filtrage, etc.

Exemple : nous prenons un exemple simple d'un système qui calcule la moyenne, le minimum et le maximum d'un grand ensemble de données. Ce système contient trois processeurs dédiés à l'exécution des trois fonctionnalités. Dans le cas de parallélisme de tâches, chacun des trois processeurs est dédié à exécuter respectivement la moyenne, le minimum et le maximum de l'ensemble de données. Les trois processeurs fonctionnent en parallèle d'une manière indépendante l'un de l'autre. Dans le cas de parallélisme de données, les trois processeurs sont dédiés d'abord à calculer la moyenne, ensuite le minimum et enfin le maximum de l'ensemble des données.

3.2.3.2 Machines parallèles

Une machine parallèle est essentiellement un ensemble de processeurs qui coopèrent et communiquent à travers des mémoires partagées ou distribuées. Michael Flynn a classifié ces machines, en fonction des flots d'instructions et de données [44]. Il les a ainsi rangées en trois grandes catégories. Nous les détaillons brièvement.

Machine SIMD : la machine SIMD, ou en anglais *Simple Instruction Multiple Data*, consiste en un ensemble de processeurs surveillés par la même unité de contrôle. Ces processeurs exécutent la même instruction en même temps, chacun sur ses propres données possédées dans sa propre mémoire locale. Ce genre de machines conduit souvent à une programmation basée sur le parallélisme de données.

Machines MIMD et SPMD : la machine MIMD, ou en anglais *Multiple Instruction Multiple Data*, consiste en un ensemble de processeurs dont chacun a sa propre mémoire contenant des données locales. Chaque processeur peut donc travailler indépendamment des autres processeurs. Habituellement, les processeurs de ces machines ne sont pas très nombreux, mais ils sont très puissants. Ce genre de machines possède deux modèles de programmations : parallélisme de données et parallélisme de tâches. Les machines MIMD fonctionnent en mode SPMD, ou en anglais *Single Programme Multiple Data* lorsqu'un même programme s'exécute sur tous les processeurs.

3.2.3.3 Exemple d'une architecture parallèle à mémoire partagée

Une architecture PRAM est représentée par un modèle abstrait de processeurs parallèles dédiés aux traitements d'applications hautes performances. Le modèle PRAM est représenté par une machine parallèle, composée de plusieurs processeurs $P_1 ... P_p$, et ayant une mémoire partagée divisée en des cellules $M_1 ... M_m$. Les processeurs fonctionnent d'une manière synchrone selon une horloge globale. Chaque processeur est considéré comme étant une machine à accès aléatoire pouvant accéder aux différentes cellules de la mémoire. Une étape de synchronisation des différents accès à la mémoire est donc nécessaire. Un processeur exécute une étape de calcul de manière atomique à chaque cycle d'horloge. Au cours d'une étape de calcul, un processeur lit des données d'une ou plusieurs cellules de la mémoire partagée, effectue une opération locale et écrit le résultat dans une ou plusieurs cellules de la mémoire partagée. Dans le but de simplifier la modélisation, nous considérons un nombre fini de processeurs.

Dans le cadre de cette thèse, nous considérons des architectures MIMD de types PRAM. Les processeurs ont le droit d'accès concurrent en lecture et en écriture sur la mémoire partagée (CRCW). Il est donc possible d'obtenir des conflits d'accès à la même cellule mémoire. Si ces accès sont en mode écriture, un écrasement, de données utiles, peut avoir lieu. Nous faisons donc l'hypothèse que l'ordre des accès à la mémoire est connu à priori et que le *mapping mémoire* est pré-calculé. Dans ces conditions, nous ne pouvons traiter que des applications ayant un comportement statique. Malgré ces hypothèses sur l'architecture, la méthodologie d'analyse que nous proposons est valable pour une large classe d'applications (traitement d'image, radar et en général tous les traitements statiques flot de données).

3.2.4 Mémoire

Les mémoires sont des composants matériels qui permettent le stockage de données pour des durées et des usages variables. Dans un système multiprocesseur on peut distinguer trois types d'organisation de mémoires. Cette distinction est basée sur le temps d'accès des processeurs aux mémoires : des accès uniformes dans le cas d'une mémoire partagée, des accès non uniformes dans le cas d'une mémoire distribuée et des accès cachés.

3.2.4.1 Mémoire partagée

Un MPSoC ayant une mémoire partagée garantie des accès équitables aux mémoires pour tous les processeurs. L'architecture de ces systèmes est basée sur l'utilisation d'une mémoire centralisée et partagée entre les différents processeurs. La mémoire centralisée, d'une part, et l'ensemble des processeurs, d'autre part, sont connectés par un simple bus

partagé ou un *crossbar switch*. Un bon exemple de telles architectures est l'architecture SMP (*Symmetric MultiProcessors*) [66].

Cette architecture est très avantageuse dans le cas d'un nombre réduit de processeurs. L'utilisateur n'a pas besoin de savoir le lieu des données ni de s'occuper des communications inter-processeurs. Cependant, le nombre de processeurs pouvant être utilisés est restreint par le problème d'accès à la mémoire. Si par exemple, on a un nombre important de processeurs dans une architecture à bus partagé, ce bus devient saturé rapidement, et par conséquent les accès à la mémoire deviennent quasi-impossibles.

3.2.4.2 Mémoire distribuée

Les MPSoC à mémoires distribuées sont la combinaison de plusieurs systèmes multiprocesseur à mémoire partagée dont les temps d'accès à la mémoire ne sont pas uniformes. Ces architectures sont proposées pour faire face aux problèmes d'accès mémoire dans les architectures à mémoire partagée, qui se traduisent par une dégradation globale des performances du système au fur et à mesure que le nombre de processeurs augmente. L'architecture ayant une mémoire distribuée consiste à préférer l'assemblage de petits modules (ensemble de processeurs) interconnectés autours d'un réseau de communication plutôt qu'un seul module contenant l'ensemble total des processeurs.

Sachant que chaque ensemble de processeurs dans un module contient un bus local, les mémoires distribuées dans chacun des modules sont quant à elles partagées et donc exploitées par tous les processeurs comme étant une seule et unique mémoire faisant partie de leur espace d'adressage. Toutefois, les accès mémoires ne sont pas égaux. En d'autres termes, le coût d'accès à un endroit précis de la mémoire distribuée est différent pour chaque processeurs par rapport aux autres, selon que la mémoire utilisée est sur le même bus local ou bien appartenant à un autre module. Comparé à une architecture à mémoire partagée, cette architecture peut utiliser des tailles mémoires très grandes. Cependant, l'efficacité des programmes sur les machines à mémoire distribuée ne peut être obtenue que par des placements appropriés de données.

3.2.4.3 Mémoire cache

Ce dernier type de mémoire est un mélange des deux types de mémoires partagées et distribuées. Une architecture ayant des mémoires caches, aussi connues sous le nom de *Cache Only Memory Access*, contient un modèle de gestion de mémoires dans un environnement multiprocesseur. Ce modèle est dérivé du modèle de mémoires distribuées où les mémoires partagées par les différents processeurs sont remplacées par des mémoires caches. L'ensemble de ces mémoires forme une mémoire globale partagée par tous les processeurs. Les données traitées sont donc susceptibles de migrer de cache en cache en fonction du processeur qui les exploite.

3.3 Techniques d'analyse

Nous présentons dans cette section des techniques d'analyse de MPSoC. Particulièrement, nous nous intéressons aux techniques d'analyse temporelle, de la gestion d'énergie, de niveaux de parallélisme potentiel et de la gestion de mémoires.

3.3.1 Analyse temporelle

Dans la problématique d'analyse temporelle de haut niveau de MPSoC, il est nécessaire de tenir compte de certaines contraintes temporelles telles que l'ordre d'exécution des différentes fonctionnalités ainsi que des optimisations faites sur les caractéristiques des architectures cibles (nombre et fréquence de processeurs, taille des mémoires, etc.). Cela aboutit à des estimations de configurations idéales de systèmes. Pour raffiner ces estimations, des analyses temporelles via des simulations de bas niveau et de prototypages sont ensuite considérées.

3.3.1.1 Analyse d'ordonnançabilité

La problématique de l'ordonnancement de tâches consiste à choisir une politique d'attribution de tâches aux processeurs qui vont les exécuter tout en garantissant certaines contraintes temporelles. Ces tâches peuvent s'exécuter à des intervalles réguliers (tâches périodiques) ou de manière aléatoire (tâches non périodiques). Quand un ordonnancement valide existe, nous disons que le système est ordonnançable.

Caractéristique d'un algorithme d'ordonnancement

Les algorithmes d'ordonnancement de tâches sur des processeurs peuvent être classifiés selon plusieurs catégories :

- **monoprocesseur/multiprocesseur** : quand une architecture ne dispose que d'un seul processeur, on dit que le problème d'ordonnancement est monoprocesseur. Quand plusieurs processeurs sont disponibles, le problème d'ordonnancement est dit multiprocesseur ;

- **préemptif/non-préemptif** : quand un ordonnancement est préemptif, l'ordonnanceur peut interrompre l'exécution d'une tâche au profit d'une autre tâche plus prioritaire. À l'inverse, un ordonnanceur ne peut pas interrompre l'exécution d'une tâche non-préemptive. Il doit donc attendre sa terminaison avant de débuter l'exécution de toute autre tâche ;

- **hors-ligne/en-ligne** : dans un ordonnancement hors-ligne (*off-line scheduling*), un calcul, préliminaire à l'exécution, permet de trouver un ordonnancement valide, fiable pour l'ensemble des tâches. Dans le cas d'un ordonnancement en-ligne (*on-line scheduling*), les décisions d'ordonnancement de tâches sont prises pendant la vie du système. L'ordonnanceur réévalue, à chaque nouvelle demande, la tâche qui doit être exécutée.

Faisabilité d'un ordonnancement

Un ordonnancement est dit faisable ou valide si toutes les tâches sont exécutées tout en respectant des contraintes temporelles imposées au préalable. Afin de vérifier la faisabilité d'un ordonnancement, avant l'exécution des tâches, l'ordonnanceur doit anticiper ces actions en assumant les pires temps d'exécutions de tâches. Les tests de faisabilité d'un ordonnancement peuvent être basés sur l'utilisation de processeurs (mesure le temps pris par un processeur pour exécuter une ou plusieurs tâches) et/ou l'analyse de temps de réponse (en calculant le pire temps d'exécution des tâches).

Par exemple, une condition nécessaire (mais pas suffisante) pour obtenir un ordonnancement valide peut être liée à l'utilisation (U) d'un processeur $Proc_i$. L'utilisation d'un processeur est calculée selon la formule suivante :

$$\sum_{i=1}^{n} \frac{C_i}{P_i} \leq U \tag{3.3}$$

où C_i et P_i représentent respectivement le temps d'exécution et la période d'une tâche T_i, n étant le nombre total de tâches impliquées dans l'exécution et U l'utilisation du processeur.

3.3.1.2 Algorithmes d'ordonnancement

Dans la littérature, plusieurs algorithmes d'ordonnancements sont définis. Parmi plusieurs, nous citons :

- **Earliest Deadline First (EDF) :** EDF est un algorithme d'ordonnancement dynamique utilisé dans les systèmes d'exploitation temps réel [73]. Les tâches sont placées dans des files de priorité. Chaque fois qu'un processeur est libre, la tâche ayant la plus petite échéance aura la priorité et sera allouée au processeur libre pour le prochain traitement. Étant donné un ensemble de tâches, EDF ordonnance toutes les tâches de telle sorte que toutes les tâches s'exécutent avant le temps d'échéance imposé. Pour une exécution mono-processeur, un ordonnancement de tâches périodiques ayant des périodes égales aux *deadline*, EDF offre une utilisation du processeur de 100 % avec $U = 1$ selon la formule 3.3.
- **Rate Monotonic (RM)** : RM est un algorithme d'ordonnancement préemptif à priorité statique utilisé dans l'ordonnancement d'un ensemble de tâches périodiques indépendantes, et à échéance sur requête [73]. Les priorités sont allouées aux différentes tâches de la façon suivante : plus la période de la tâche est courte, plus sa priorité sera importante. La condition suffisante d'ordonnançabilité, selon RM, pour un système composé de n tâches est $U = n(2^{\frac{1}{n}} - 1)$, selon la formule 3.3.
- **Deadline Monotonic (DM) :** DM un algorithme d'ordonnancement à priorité statique. La priorité d'une tâche est inversement proportionnelle à son échéance. Cet algorithme est équivaut à l'algorithme Rate Monotonic quand l'échéance d'une tâche est égale à sa période.

3.3.1.3 Analyse de synchronisation à l'aide d'horloges abstraites

Les langages synchrones flots de données (par exemple Lustre, Signal, etc.) permettent la spécification de réseaux de processus communicants d'une manière synchrone. Dans ce genre de programmation, chaque flot de données est associé à un type d'horloges logiques, qui indique les instants de présence de valeurs sur le flot.

Horloges affines de Signal

Les horloges affines ont été définies comme étant une extension de la notion d'horloge habituelle du langage Signal. Elles permettent la modélisation et la vérification de systèmes temps réel, non nécessairement synchrone, avec des outils synchrones [111]. Dans un système d'horloges, les instants deux horloges clk_i et clk_j distinctes peuvent avoir un comportement affine l'un par rapport à l'autre. Quand cela est le cas, les

instants de l'horloge clk_j peuvent être tracés à partir des instants de l'horloge clk_i selon une transformation affine.

Definition 6 (Transformation affine) *Une transformation affine, ayant les paramètres* (n, ϕ, d)*, appliquée à une horloge* clk_1 *produit une horloge* clk_2 *par l'insertion de* $(n-1)$ *instants entre deux instants successifs de* clk_1*, et ensuite compter sur cette série fictif d'instants chaque* $d^{ième}$ *instant, en commençant par le* $\phi^{ième}$ *instant. Les horloges* clk_1 *et* clk_2 *sont dit en relation affine* (n, ϕ, d)*, noté aussi* $clk_1 \overset{(n,\phi,d)}{\rightarrow} clk_2$ *(voir l'exemple dans la Figure 3.1).*

	0	1	2	3	4	5	6	7	8	9	10	...
clk_1:	tt	⊥	⊥	tt	⊥	⊥	tt	⊥	⊥	tt	⊥	...
clk_2:	⊥	tt	⊥	⊥	⊥	tt	⊥	⊥	⊥	tt	⊥	...

Figure 3.1: Trace de deux horloges clk_1 et clk_2 en relation $(3, 1, 4)$-affine.

Dans les systèmes contenant des horloges affines, deux signaux différents sont dits synchronisable s'il y a un moyen, à base de flots de données, de les rendre réellement synchrone. Le compilateur Signal a été étendu afin de pouvoir intégrer, dans le calcul des relations d'horloges, une phase d'analyse statique qui résout les questions concernant la synchronisation des horloges.

Horloges k-périodiques

Les réseaux de Kahn N-synchrones, définis par Cohen et al. [29], s'inspire du modèle synchrone. Ils permettent de décrire un comportement (ultimement k) périodique de systèmes. Des horloges k-périodiques, fonctionnant avec les opérateurs *when* et *merge*, sont définies et associées à différents composants d'un système.

Une étude de synchronisation de composants, par le biais de ces horloges, permet de vérifier certaines contraintes de synchronisation. Quand l'analyse d'horloges assure la possibilité de synchronisation, la solution consiste à insérer des mémoires tampons de taille finie afin de stocker les données générées et non pas encore consommées. En insérant du code représentant des mémoires tampons, il devient possible de modéliser des systèmes non nécessairement synchrones avec des outils synchrones.

Les horloges k-périodiques permettent de capturer une exécution périodique d'un système. Ils reposent sur une syntaxe binaire dont l'occurrence de 1 signifie que l'horloge est présente et l'occurrence de 0 indique son absence. La grammaire est définie comme suit :

- un mot (ou *word* en anglais) définit un comportement périodique et est défini par : $mot = offset(period)$.
- le $offset$ reflète le comportement initial avant le commencement du comportement périodique. Cette valeur est représentée par une séquence de valeurs binaires finies : $offset = [0|1]^*$.
- la période représente le comportement périodique du système. Elle est représentée par une séquence de valeurs binaires finies et peut être répétée infiniment. Sa définition est la suivante : $period = [0|1]^+$.

Plusieurs fonctions sont définies permettant de manipuler des périodes. Nous citons quelques-unes et nous montrons une application de chacune des fonctions sur l'exemple d'une période $p = 001011010$:

- **Longueur** : elle calcule le nombre de valeur 0 ou 1 d'un mot binaire (que ça soit une horloge k-périodique ou une période). Elle est notée $|period|$ (par exemple $|p| = 9$) ;
- **Nb_occurrence** : elle compte le nombre d'occurrence d'une valeur binaire. Pour une occurrence d'une valeur 1 et 0, cette fonction est notée respectivement $|period|_1$ et $|period|_0$ (par exemple $|p|_1 = 4$ et $|p|_0 = 5$) ;
- **Position** : elle détermine la position d'une valeur 1 dans un mot binaire. Elle est notée $[period]_k$ dénotant la position du $k^{\text{ième}}$ occurrence de la valeur 1 sur le mot *period* (par exemple $[p]_3 = 6$).

Cependant, l'analyse de synchronisation s'avère coûteuse si les traces d'horloges sont longues et ne peuvent être appliquées que sur du comportement périodique. Par conséquence, les auteurs étendent la notion d'horloges binaires infinies par le concept d'*enveloppe* afin de permettre la spécification de comportements apériodiques [30]. Une *enveloppe* représente une abstraction d'une horloge k-périodique. C'est un simple moyen de description permettant de raisonner en moyenne sur des horloges k-périodiques.

3.3.2 Gestion d'énergie

La gestion de puissance d'un MPSoC se réalise selon différentes manières. Par exemple, des mises en veille de certaines parties d'un circuit ou des changements de couples de valeurs (tension, fréquence) permettent de réduire considérablement la puissance dissipée.

En effet, certaines applications ont des comportements réguliers (exécution de tâches périodiques par exemple) et sont soumises à des contraintes temporelles (par exemple temps d'échéance de l'exécution d'une fonctionnalité). Ces informations peuvent être exploitées afin de gérer la consommation d'énergie globale du système. Par exemple, en diminuant les couples de valeurs (fréquence, tension) pour un processeur, la puissance dissipée tend aussi à diminuer. Nous détaillons par la suite deux techniques bien connues dans la littérature : la mise en veille de processeurs inactifs (*power shutdown*) et l'adaptation de la tension d'alimentation et de la fréquence d'un processeur (*Dynamic Voltage Frequency Scaling–DVFS*)[13].

Le fonctionnement d'un processeur n'est pas toujours constant. Il comporte des périodes d'inactivité pendant lesquelles il n'effectue aucun traitement utile pour le système (exécution de la tâche *idle* par exemple) mais continue à consommer inutilement de l'énergie. Mettre en veille le processeur pendant ces périodes permettrait d'éviter cette perte. Une autre technique efficace pour économiser de l'énergie est d'adapter la puissance de traitement du processeur aux besoins de l'application [43]. Nous pouvons considérer un ajustement conjoint de la tension et de la fréquence des processeurs afin de lisser l'utilisation des processeurs sur l'ensemble du temps d'exécution nominal (temps d'échéance). Cette diminution de la fréquence/tension aura comme effet de diminuer la puissance de traitements des différents processeurs. Cela a un impact direct sur la diminution de la consommation d'énergie totale.

3.3.3 Exploration de niveaux de parallélisme

L'exploration de l'espace de conception peut s'effectuer à plusieurs niveaux d'abstraction (par exemple niveau système, architectural, logique). L'exploration du parallélisme s'avère cruciale quand des applications hautes performances et des architectures multiprocesseur sont considérées. Cependant, la question qui se pose est la suivante :

comment gérer la mise en œuvre des applications haute performance sur des centaines de processeurs potentiellement hétérogènes et massivement parallèles?

En effet, les systèmes MPSoC ayant des calculs hautes performances peuvent impliquer des centaines de processeurs reliés par un bus en cas de mémoire partagée ou un réseau de communication dans le cas des systèmes distribués. Ces infrastructures fournissent une grande capacité de calcul. Cependant, leur complexité, leur hétérogénéité, leur dispersion géographique sur la puce et leur partage des applications concurrentes posent un problème majeur. L'exploitation de cette infrastructure présente de nombreuses problématiques qui restent ouvertes, parmi lesquelles :

- comment fixer le nombre de processeurs ?
- quel type d'association tâches/processeurs choisir pour garantir de meilleures performances à moindres coûts ?
- comment éviter la congestion de bus dans des architectures à mémoire partagée ?

Les critères, cités ci-dessus sous forme de questions, affectent gravement les performances de systèmes. Bien que l'avancement de la technologie des semi-conducteurs a permis une intégration de centaines (voir des milliers) de processeurs et de co-processeurs, il est préférable d'utiliser juste le nécessaire pour assurer le bon fonctionnement de systèmes avec des tolérances aux pannes. Une mauvaise estimation de ressources physiques (processeurs, co-processeurs, mémoires) peut induire des coûts économiques énormes. De plus, il existe un grand nombre de choix possibles de configurations d'associations de tâches fonctionnelles sur des processeurs. Plus le nombre de processeurs est grand et plus les fonctionnalités sont complexes, le nombre d'associations possibles augmente exponentiellement. Il devient donc impossible de tester et de simuler l'espace de solutions avec des outils d'analyse et d'exploration de bas niveaux (par exemple RTL). Enfin, l'augmentation du nombre de processeurs, dans une architecture multiprocesseur à mémoire partagée, a des conséquences directes, et parfois graves, sur les performances du système en question. Cela est dû à la congestion de bus quand un grand nombre de processeurs sont considérés.

3.3.4 Gestion mémoire

La gestion de mémoire a suscité une grande attention pour les concepteurs de MPSoC. Cela est dû à son impact fort sur la durée de calculs effectués par l'ensemble des processeurs impliqués dans l'exécution de fonctionnalités. L'exécution d'une simple instruction par exemple requiert plusieurs accès mémoire. Cela impose un temps de latence pour accéder à la mémoire. Ce temps peut être représenté par le nombre de cycles d'horloge entre l'envoi de la requête à la mémoire (lecture ou écriture) et quand l'opération sur la mémoire à terminer. Quand l'accès à une mémoire est réservé à un seul processeur (par exemple l'accès à une mémoire cache d'un processeur), le temps d'accès est toujours pareil en cas de disponibilité de données. Cependant, en cas d'absence de données dans une mémoire cache, le processeur accède à une mémoire partagée lointaine. Bien évidement, cet accès est plus coûteux en terme de temps d'attente du processeur.

Le temps d'accès à une mémoire peut donc varier d'un accès à un autre. Il dépend fortement du type, de la taille et de l'emplacement de la mémoire considérée. En effet, plus la taille d'une mémoire est grande et son emplacement par rapport aux processeurs est loin, plus l'accès aux données est lent. Cette emblématique a imposé une structuration

hiérarchique de mémoires afin d'améliorer les performances de systèmes. Les mémoires sont d'autant plus petites et rapides qu'elles sont proches des processeurs alors que les mémoires de grandes capacités et un temps d'accès lent sont plus loin.

Afin d'éviter l'accès fréquent à des mémoires lentes (mémoire partagée de grande taille et lointaine), des instructions et des données sont dupliquées dans dés mémoires caches qui sont plus rapides et de petite taille. La requête sur une donnée est d'abord adressée au cache. On dit qu'il y a un succès de cache (*cache hit*) si le cache peut satisfaire la requête, sinon, il s'agit d'un défaut de cache (*cache miss*) qui désigne un accès à une donnée ne se trouvant pas dans le cache, ou plus généralement, n'étant pas valide en cache.

Une architecture multiprocesseur multi-caches est composée d'un ensemble de processeurs dont chacun possède son propre cache. Par conséquent, une donnée peut se trouver dans plusieurs caches simultanément. En vue d'assurer la cohérence entre les données présentes dans les différents caches, les accès sont orchestrés par des protocoles de cohérences de caches.

Dans la section suivante, nous présentons un état de l'art récent sur des travaux abordant l'analyse de MPSoC. Notamment, nous nous intéressons aux travaux abordant l'analyse de synchronisation et d'ordonnancement de tâches, à l'évaluation et la diminution de la consommation d'énergie et enfin aux environnements d'exploration de niveaux de parallélisme. Ces travaux sont particulièrement décrits et analysés afin de souligner clairement les différentes contributions de cette thèse.

3.4 Quelques travaux existants

Analyse de synchronisation

Les approches d'analyses de synchronisation de composants sont nombreuses dans la littérature [108, 137, 55, 79, 20, 29, 111]. Celles qui sont faites à haut niveau nous intéressent le plus. Pour cela, nous ne présentons que les travaux qui abordent la synchronisation de composants en se servant des horloges affines et k-périodiques. Cela nous permet de positionner nos travaux par rapport à eux :

- `Glitia et al.` [55] proposent une modélisation et une analyse de modèles SDF par le biais d'horloges logiques et de contraintes d'horloges. Un graphe de tâches SDF est composé d'un ensemble de tâches, appelées acteurs, qui communiquent à travers des connecteurs, appelés arcs. Ils utilisent le langage CCSL (*Clock Constraint Specification Language*) qui permet la modélisation de contraintes synchrones et/ou asynchrones sur des couples d'horloges. Les acteurs sont modélisés en tant que horloges logiques alors que les arcs sont modélisés en tant que contraintes d'horloges. Un instant logique d'une horloge indique l'activation de l'acteur associé à l'horloge. Les auteurs définissent ensuite un algorithme d'ordonnancement de tâches pour des pairs d'acteurs ayant des communications directes. Ainsi, les dépendances de données vont donner un ordre d'exécution. La motivation de ce travail est d'ajouter, dans un graphe de tâches flots de données, des informations supplémentaires concernant l'exécution d'un système (par exemple contraintes d'environnement). Rendre explicite ces contraintes dans un modèle SDF permet une exploration de nouvelles possibilités d'ordonnancements possibles.

- `Metivier et al.` [79] présentent une extension du modèle de calcul polychrone de Signal avec une notation d'horloges exprimant des relations de périodicités entre des signaux. La principale motivation de leur démarche est d'analyser

des communications inter-processus afin d'évaluer des tailles minimales de mémoires tampons nécessaire pour avoir une exécution correcte de systèmes. Ce travail est inspiré des réseaux de Kahn n-synchrones [29]. Cependant, ils étendent la notation binaire des horloges k-périodiques en une notation ternaire, dénotée par *atome*, décrivant les signaux d'horloges.

- `Boucaron et al.` [20] empruntent l'hypothèse des réseaux de Kahn N-synchrone pour décrire formellement les stations de relais et les *shell wrappers* sur des circuits synchrones. Le modèle de départ est un graphe CNS (*Computation Network Scheme*) de nœuds connectés par des arcs. Les nœuds représentent la partie calcul alors que les arcs définissent des canaux de transfert de données. Les stations de relais représentent des mémoires tampons entre deux nœuds communiquant alors que les *shell wrappers* ont pour but de calculer le temps d'activation des processus. En insérant des propriétés de corrections locales, l'analyse permet d'assurer la validation des propriétés globales du réseau. D'une part, l'analyse temporelle permet de régulariser la circulation des signaux entre les différents processus communiquant. D'autre part, un contrôle sur le flot des signaux permet d'éviter la congestion des données. Dans la description du modèle, un bloc de traitement peut être interrompu temporairement à cause de l'indisponibilité des signaux d'entrées, ou en raison de l'incapacité du réseau de mémoriser les données de sortie. Aussi, l'utilisation des éléments synchrones permet un redimensionnement des mémoires tampons entre les processus synchrones locaux, provoquant ainsi un bon fonctionnement du système.

Positionnement

Nous partageons avec Glitia et al. l'utilisation d'horloges logiques et de contraintes d'horloges CCSL dans le but d'évaluer des configurations de systèmes. Cependant nous considérons le paquetage RSM de Marte comme modèle de calcul et une co-modélisation de système selon le modèle Y. Pour cela, nous considérons des horloges purement fonctionnelles pour l'abstraction de l'application, des horloges physiques pour décrire la vitesse des processeurs et des horloges d'exécutions simulant l'activité des différents processeurs impliqués dans l'exécution.

Nous partageons avec Metivier et al. et Boucaron et al. l'utilisation des horloges k-périodiques. Nous avons associé des horloges k-périodiques à des tâches fonctionnelles. Ces horloges nous permettent d'analyser la synchronisabilité de tâches communicantes dans un graphe de tâches acyclique. En cas de possibilité de synchronisation, nous estimons des tailles minimales de mémoires tampons nécessaires pour la sauvegarde de données. Cependant, nous nous distinguons de ces travaux par la prise en considération, dans l'analyse temporelle, des architectures multiprocesseur et des associations entre des applications et des architectures. Nous adoptons en cela une analyse temporelle, basée sur le modèle Y, d'applications hautes performances exécutées sur des architectures parallèles. En effet, nous prenons en compte, non seulement les communications inter-tâches d'une application, mais aussi des processeurs qui sont chargés l'exécution de fonctionnalités. La prise en compte des processeurs dans l'analyse temporelle permet de détecter des violations de l'ordre d'exécution de tâches. Nous comblons ainsi le fossé qui existe entre, d'une part, une spécification d'un graphe de tâches considéré comme étant synchrone, et d'autre part, son exécution sur une architecture considérée comme étant potentiellement asynchrone.

Travaux	MoC	Horloges	Objectifs	Support d'analyse	Propositions
Glitia et al.	SDF	logiques	ordre d'exécution correcte de tâches fonctionnelles	CCSL	analyse temporelle de modèles SDF avec CCSL
Metivier et al.	Polychrony	logiques	dimensionnement de mémoires tampons	réseaux de kahn N-synchrone	analyse de communication inter-processus
Boucaron et al.	CNS	logiques	régularisation de la circulation de signaux, dimensionnement de mémoires	réseaux de kahn N-synchrone	analyse de transmission de signaux
Contribution de cette thèse	Marte	logiques	ordre d'exécution correcte de tâches, estimation de temps d'exécution, estimation de la consommation d'énergie	réseaux de kahn N-synchrone	analyse de l'exécution d'applications hautes performances sur des architectures massivement parallèles

Table 3.1: Comparaison de travaux abordant l'analyse de synchronisation de composants fonctionnels.

Techniques d'ordonnancement de tâches

Nous présentons dans cette section des travaux récents abordant l'ordonnancement de tâches. Cet état de l'art nous est utile afin de se positionner par rapport aux travaux récents. Nous citons les travaux suivants :

- `Forget et al.` [45] présentent un langage formel de programmation de systèmes temps réel. Ils considèrent un système comme étant un assemblage de plusieurs systèmes synchrones, localement mono-périodiques. L'ensemble des systèmes synchrone représente un système global multi-périodique. Le langage permet de préciser des schémas de communication déterministes complexes entre plusieurs fonctions externes écrites dans un langage de programmation (comme du C ou du Lustre). À partir d'une spécification fonctionnelle, ils obtiennent des tâches périodiques ayant des contraintes de précédences et de périodicités. Une fois le système programmé, il est compilé en un ensemble de tâches temps réel implantées sous forme de threads C communicants. Le code généré s'exécute à l'aide d'un système d'exploitation classique disposant de la politique d'ordonnancement EDF. Les opérations du programme s'exécutent de manière concurrente et peuvent donc être préemptées en cours d'exécution en faveur d'une opération plus urgente. Le protocole de communication proposé permet d'assurer que, malgré les préemptions, l'exécution du programme généré est prédictible et correspond exactement à la sémantique formelle du programme d'origine. Cependant, leur démarche n'est valide que pour le cas d'un ordonnancement monocœur.

- `Abdellatif et al.` [118] présentent une implémentation générale des systèmes temps réel en se basant sur deux modèles. Le premier modèle, du type abstrait, décrit le comportement des logiciels temps réel par un automate temporisé permettant de modéliser des contraintes temporelles indépendantes de la plateforme d'exécution. Le modèle abstrait décrit le comportement dynamique des fonctionnalités comme un ensemble de tâches interagissantes sans aucune restriction sur leurs types (périodique, sporadique, etc.). Le deuxième modèle, du type physique, décrit le comportement des logiciels temps réel durant leurs exécutions sur une plateforme physique en considérant le pire temps d'exécution des tâches. Ce modèle facilitera la vérification de certaines propriétés essentielles du modèle abstrait telles que la cohérence des contraintes temporelles et l'absence de *deadlock*. À partir de ces deux modèles, les auteurs proposent un ordonnancement valide des fonctionnalités sur les ressources physiques afin de respecter les contraintes temporelles.

- `Viehl et al.` [126] proposent des méthodes d'analyse formelle pour la conception sûre des applications embarquées critiques ayant des contraintes temporelles et exécutées sur des architectures multiprocesseur. L'accès aux ressources physiques se fait à travers des techniques d'ordonnancement à base du *time slicing*. L'approche proposée commence par la spécification des fonctionnalités en C/C++ ou SystemC (traduit ensuite en un graphe de tâches cycliques). Ensuite, une analyse de synchronisation est faite sur les composants communiquant afin d'indiquer le temps d'attente de chaque tâche avant son exécution.

- `Zhu et al.` [138] proposent des techniques pour optimiser la consommation d'énergie de systèmes. Cela est fait en analysant les valeurs de fréquences et de tensions. Ces techniques sont appliquées sur l'algorithme d'ordonnancement *Earliest Deadline First* (EDF). L'analyse ajuste la vitesse de l'exécution des tâches

dynamiquement selon des statistiques faites sur le workload potentiel. Cet ajustage doit garantir l'exécution des tâches avant leur temps d'échéance.

- `Jung et al.` [63] présentent un algorithme d'ordonnancement qui divise les tâches périodiques applicatives en des sous tâches en utilisant une technique de partage de tâches. Dans la méthodologie proposée, les sous tâches résultant du partage sont ordonnancées sur des architectures multiprocesseur en assurant leur exécution avant la date d'échéance. Chaque processeur est caractérisé par un seul paramètre reflétant sa puissance de calcul. L'algorithme d'ordonnancement Rate Monotonic (RM) est utilisé afin d'ordonnancer chaque sous tâche sur son propre processeur. Le but ultime est d'assurer l'utilisation maximale des processeurs dans une architecture multiprocesseur.

Positionnement

Dans le cadre de cette thèse, nous proposons un algorithme d'ordonnancement, du type statique non préemptif, de fonctionnalités sur des ressources physiques. Cet ordonnancement est représenté par trois types d'horloges : fonctionnelle, physique et exécution. Les horloges fonctionnelles décrivent les comportements temporels des fonctionnalités tout en préservant les contraintes temporelles qui peuvent être imposées (périodicité ou précédence). Les horloges physiques permettent de décrire les vitesses d'exécution de processeurs. Les horloges d'exécution, à leurs tours, décrivent le comportement temporel des fonctionnalités quand ils sont exécutés sur une architecture. Cette séparation de vue entre la description du comportement fonctionnel et des contraintes temporelles d'une part et la description de l'exécution, d'autre part, ressemble beaucoup à celle adoptée par Abdellatif et al. Afin de partager les ressources physiques d'une architecture multiprocesseur, nous utilisons la technique d'ordonnancement *time slicing* comme dans les travaux de Viehl et al. Nous partagons avec Zhu et al. et Jung et al. l'objectif de maximiser l'utilisation de processeurs. En effet, nous analysons statiquement la quantité de travail (ou *worklaod*) de chaque processeur. En fixant une durée maximale pour l'exécution de fonctionnalités, nous pouvons déduire une fréquence minimale pour chaque processeur. Ces valeurs de fréquences assurent l'utilisation maximale de la puissance du processeur durant l'exécution des tâches fonctionnelles qui lui sont associées. Cela doit être fait tout en garantissant l'exécution des tâches avant leur temps d'échéance comme dans Yifan et al. [138].

Consommation d'énergie

- `Seo et al.` [106] proposent un environnement d'estimation d'énergie selon le style d'architecture (clients-serveurs, *publish-subscribe*, *pipe-and-filter*, *peer-to-peer* etc.) dans les systèmes distribués. Ces estimations, visant la consommation d'énergie du système, permettent de choisir le style d'architecture le plus approprié pour une application distribuée donnée. Le coût énergétique d'un composant du système est considéré comme étant la somme des coûts d'énergie de la partie qui fait le calcul et de la communication représentée par un échange de données via des connecteurs. L'utilité de cette analyse est son application sur différents styles d'architecture pour l'évaluer en utilisant un outil intermédiaire qui permet par la suite une implémentation réelle des différentes architectures.

- `Watanabe et al.` [128] se concentre sur l'ordonnancement des applications périodiques qui ont à la fois des contraintes de latences de calcul et de débit.

Travaux	Proposition	Vérifications	Algorithmes	Architectures	Applications
Forget et al.	langage formelle d'ordonnancement de tâches	contraintes de périodicité et de précédence	EDF	mono-proc.	Temps réel
Abdellatif et al.	modèle abstrait fonctionnel, modèle abstrait physique	absence de *deadlock*, cohérence de contraintes temporelles	algorithme développé en interne	mono-proc.	Temps réel
Viehl et al.	méthodes d'analyse formelle	contraintes temporelles	algorithme développé en interne (*time slicing*)	multiproc.	critiques
Zhu et al.	techniques d'analyse	ajustage de fréquences et de tensions de processeurs	EDF	multiproc.	Temps réel
Jung et al.	algorithme de partage de tâches	utilisation maximale de processeurs	RM	multiproc.	Temps réel
Contribution de cette thèse	méthodologie d'analyse temporelle	contraintes de périodicité et de précédence, ajustage de fréquences et de tensions de processeurs	algorithme développé en interne	multiproc.	hautes performances

Table 3.2: Comparaison de travaux abordant l'ordonnancement de tâches.

L'ensemble des instances d'application sont représentées sous forme d'un graphe de tâches. Un ordonnancement statique pipeliné de tâches est défini afin de minimiser autant que possible la consommation d'énergie des systèmes MPSoC de types *Globally Asynchronous Locally Synchronous* (GALS) soumis à des contraintes de performances. En effet, dans la méthodologie proposée, des applications de types GALS sont modélisées ainsi que des architectures MPSoC. Ensuite, l'algorithme d'ordonnancement, basé sur la technique recuit simulé est appliqué afin de trouver l'ordonnancement optimal. La solution optimale de cette approche est celle qui choisit les meilleurs choix de la tension/fréquence des processeurs afin de l'utiliser ultérieurement dans des algorithmes d'ordonnancement dynamiques. Les coûts de la communication inter-processus en terme de performance et d'énergie sont pris en compte dans l'analyse.

- `Bautista et al.` [11] définissent un algorithme heuristique, basé sur un ordonnanceur strict, pour distribuer équitablement la quantité de travail dans un système multiprocesseur. L'ordonnanceur, qui prend en compte la consommation d'énergie, est composé d'un algorithme d'ordonnancement EDF pour chaque processeur et d'un contrôleur de la tension. La vitesse minimale du processeur est choisie par l'ordonnanceur de telle sorte que l'exécution des tâches respecte les contraintes d'ordonnancement imposées par EDF. Cette information est envoyée au contrôleur de la tension. Une fois que ce dernier reçoit les vitesses de tous les ordonnanceurs EDF du système, il fournit aux processeurs la valeur de fréquence/tension appropriée afin de satisfaire toutes les contraintes telles que le respect du temps d'échéance. Concernant la distribution de la quantité de travail, un partitionnement est appliqué sur l'ensemble des tâches avec les CPU ou les mémoires de telle manière à augmenter l'imbrication du temps d'exécution. Cela réduira le temps d'exécution. Le temps gagné pourra être ainsi consacré à la réduction de la consommation d'énergie. La quantité de travail sera enfin distribuée d'une manière équitable pour tous les processeurs.

- `Grosse et al.` [56] proposent une méthode d'optimisation de l'énergie total d'un système. Cette méthode est applicable sur des systèmes de flots de données sur puce. Elle peut être aussi appliquée sur des systèmes distribués ayant des scénarios d'exécution connus en avance. La méthode consiste à optimiser les fréquences de chaque unité fonctionnelle selon les contraintes globales imposées sur le scénario en terme de latence et de débit. La méthode est particulièrement adaptée aux systèmes qui considèrent les pires conditions d'exécution comme c'est le cas des systèmes de télécommunications data-intensive. L'objectif final est de réduire l'énergie pour les nœuds sans altérer le comportement fonctionnel. Cette réduction se fait par le réglage de la valeur des fréquences/tensions et un redimensionnement des mémoires tampons de types FIFO dédiées à la synchronisation des différents éléments de calculs du système. La méthodologie proposée est facilement applicable sur des MPSoC complexes. Les coûts sont faibles en terme de mémoire et de CPU car la valeur des fréquences est calculée hors-ligne pour chaque configuration.

Environnement d'exploration de niveaux de parallélisme

Une exploration intelligente du niveau du parallélisme de MPSoC hautes performances est primordiale afin d'aboutir à de meilleures performances. Nous présentons dans cette

Travaux	Applications	Types	Propositions
Seo et al.	*peer-to-peer*, client-serveur, *pipe-and-filter*	dynamique	environnement d'estimation d'énergie pour différent type d'architecture
Watanabe et al.	MPSoC périodique, contrainte de latence et de bande passante	statique	ordonnancement optimal de tâches vis-à-vis de la consommation d'énergie
Bautista et al.	MPSoC	dynamique	ordonnancement EDF et contrôleur de valeur fréquence/tension de processeurs
Grosse et al.	flot de données	statique	optimisation des valeurs de fréquences/ tensions de processeurs, redimensionnement de mémoires tampons

Table 3.3: Comparaison de travaux abordant la consommation d'énergie.

section quelques travaux abordant l'exploration des niveaux de parallélisme dans la conception de MPSoC :

- Ristau et al. [101] présentent une stratégie d'estimation de performance de MPSoC, basée sur une méthodologie d'analyse de parallélisme de tâches. L'objectif est d'évaluer un grand nombre de configuration possible rapidement. Cette analyse permet de déterminer le nombre idéal de processeurs pour répondre aux contraintes de performances et d'énergies. Une topologie de réseaux de communication est fixée afin d'évaluer les transferts de données entre différents processeurs.

- Cordes et al. [31] proposent un algorithme qui délimite la granularité des tâches fonctionnelles selon les exigences architecturales et le temps d'exécution global. Le point d'entrée est un graphe de tâches hiérarchique. Ils utilisent la programmation linéaire (*integer linear programming–ILP*) afin d'étudier un parallélisme potentiel de tâches à chaque niveau d'hiérarchie. Cela permet de trouver les meilleures distributions de tâches fonctionnelles aux processeurs disponibles.

- Yang et al. [135] définissent un algorithme d'allocation et d'ordonnancement de tâches pour des architectures MPSoC. Afin de profiter au maximum des processeurs dans une architecture, les auteurs considèrent un parallélisme de tâches, de données et temporels. En exploitant les trois niveaux de parallélisme, de meilleurs performances sont obtenues pour le même nombre de processeurs.

  Le parallélisme de tâches consiste à ordonnancer des tâches fonctionnelles sur des processeurs. Un surcoût de communication inter-processeurs est pris en compte. Quant au parallélisme de données, des ensembles de données d'entrée pour une tâche fonctionnelle sont exécutés sur différents processeurs. Le parallélisme temporel peut être obtenu en appliquant un pipeline sur le graphe de tâches. En divisant un graphe de tâches en plusieurs étapes pipelines, une itération courante d'une tâche peut être exécutée en même temps que l'itération précédente.

3.5 Contexte de l'analyse proposée dans cette thèse

Dans le cadre de cette thèse, nous proposons une méthodologie d'analyse et de vérification, au niveau système, de modèles MPSoC permettant une estimation de performance à la fois rapide et précise. L'idée sous-jacente de notre méthodologie peut être divisée en plusieurs parties. Nous abstrayons un modèle MPSoC, co-modélisé à haut niveau, par le biais d'horloges abstraites. Ces dernières nous servirons comme support d'analyse.

Afin de pouvoir analyser un système et vérifier des contraintes fonctionnelles et non fonctionnelles, nous avons besoin des informations suivantes :

- la quantité de travail au niveau application, fournie sous forme d'activations de tâches ;
- l'ordre d'exécution de tâches ;
- le nombre de processeurs ;
- les fréquences de processeurs (optionnelle) ;
- une association de tâches fonctionnelles sur des processeurs.

Des horloges fonctionnelles et physiques sont obtenues en analysant respectivement des modèles d'applications et d'architecture. Afin d'obtenir des horloges simulant les activités des différents processeurs durant l'exécution de fonctionnalités, nous avons défini un algorithme d'ordonnancement de type statique non préemptif. Cet algorithme projette des horloges fonctionnelles sur des horloges physiques aboutissant à une trace d'horloges d'exécutions. Nous reposons sur ces horloges afin d'analyser les contraintes suivantes :

- le respect de l'ordre d'exécution de tâches ;
- dimensionnement de mémoires ;
- l'estimation de temps d'exécution ;
- l'estimation de la consommation d'énergie totale.

En appliquant une analyse d'horloges abstraites sur l'ensemble de configurations possibles, nous arrivons à réduire cet espace en conservant seulement celles qui vérifient les critères imposés par le concepteur (faible consommation d'énergie, un nombre réduit de processeurs, etc.). Les configurations retenues seront ultérieurement analyser par des techniques d'analyses offertes par l'environnement Gaspard2 (par exemple simulation en SystemC ou prototypage en VHDL).

3.6 Synthèse

Nous avons évoqué dans ce chapitre la grande importance de la vérification des propriétés fonctionnelles (telles que la synchronisation des composants) et non fonctionnelles (telles que la diminution de la consommation d'énergie) dans le flot de conception conjointe matérielle/logicielle de MPSoC. Le problème majeur à résoudre actuellement dans le flot de conception des systèmes multiprocesseur MPSoC consiste à trouver les meilleures configurations d'architectures relatives aux découpages fonctionnels associés.

Les ordonnancements dynamiques sont généralement très précis. Cependant, ils nécessitent beaucoup de temps pour leurs développements et leurs simulations. Cela les rend, en pratique, inefficaces quand le nombre de composants fonctionnels/architecturaux est important. En contrepartie, les ordonnancements statiques ont l'avantage d'être plus rapide et facile à implémenter. Cependant, les résultats obtenus, en analysant statiquement des modèles, ne sont pas aussi précis que les approches dynamiques. Cela est dû au manque de détails d'exécution pour les approches statiques.

Chapter 4

Gaspard2: un environnement de conception conjointe pour MPSoC

4.1 Introduction

Dans ce chapitre, nous allons présenter les principes et la sémantique de Gaspard2, l'environnement que nous utilisons. La section 4.2 décrit le domaine d'application de Gaspard2, suivie par une présentation de la sémantique du langage pour l'expression du parallélisme massif d'applications et d'architectures dans la section 4.3 ainsi que les différents langages cibles. L'analyse et la vérification dans Gaspard2 sont décrites dans la section 4.4. Dans la section 4.5, nous introduisons la contribution de cette thèse dans Gaspard2.

4.2 Domaine d'application : traitement intensif de données

L'environnement Gaspard2 est dédié aux traitements d'applications hautes performances comportant du calcul intensif. Ce type d'applications est prédominant dans de nombreux domaines, tels que le traitement d'images et de vidéos ou par exemple dans les systèmes de détection de mouvement (radar, sonar). Ces applications manipulent principalement des structures de données multidimensionnelles, représentées

habituellement par des tableaux. Une image tridimensionnelle, par exemple, contient deux dimensions spatiales et une temporelle [54]. Une application sonar est un bon exemple aussi. Elle contient une dimension représentant l'échantillonnage temporel des échos, une autre dimension pour l'énumération des hydrophones. Le traitement de ce type d'applications présente un certain nombre de difficultés. Nous en citons quelques-unes :

- les applications traitées sont généralement massivement parallèles. Afin de bénéficier du parallélisme sous-jacent, on doit pouvoir l'exprimer complètement ;
- seuls très peu de modèles de calcul (MoC) sont multidimensionnels et aucun n'est largement utilisé ;
- les motifs extraits des tableaux de données sont divers et complexes ;
- l'ordonnancement de ces applications avec des architectures massivement parallèles est un grand défi dû au grand espace de solutions possibles.

4.3 Abstractions dans la conception

Dans cette section, nous présentons les différentes étapes de modélisation de MPSoc, aboutissant à une génération de codes pour différentes technologies (Lustre, Signal, SystemC, OpenMP Fortran, VHDL). D'abord nous détaillons la sémantique des différents niveaux supérieurs permettant la modélisation de systèmes. Ensuite, nous présentons les différentes technologies cibles de l'environnement Gaspard2. Le passage, des niveaux supérieurs vers les niveaux cibles, est automatique via des techniques de transformations de modèles. Enfin, nous montrons des exemples de modélisation de MPSoC hautes performances.

4.3.1 Niveaux supérieurs

L'environnement Gaspard2 suit l'approche de conception selon le modèle Y. Il se sert du profil Marte [89, 100] (*Modeling and Analysis of Real-Time and Embedded systems*) pour la spécification d'applications, d'architectures et des associations entre des applications et des architectures. Le déploiement des IP logicielles et matérielles peut être aussi modélisé en Gaspard2.

La Figure 4.1 présente l'architecture globale du profil Marte selon une décomposition en paquetages. Marte est structuré selon deux directions, la modélisation des concepts des systèmes embarqués temps réel et l'annotation des modèles d'applications pour supporter l'analyse des propriétés de systèmes. L'organisation du profil reflète cette structure, par la séparation des deux paquetages *MARTE design model* et *MARTE analysis model*.

La spécification de Marte est composée de quatre paquetages principaux :

1. `Foundations` : ce paquetage définit des notions fondamentales dans le domaine de l'embarqué. Il fournit des modèles de construction pour spécifier des propriétés non fonctionnelles (*Non-Functional Properties*–`NFPs`). Il introduit des notions de temps (`Time`), de ressources abstraites (*Generic Resource Modeling*–`GRM`) nécessaires à la modélisation de plateformes générales dédiées à l'exécution des systèmes embarqués temps réel. Le sous paquetage (*Generic Component Model*–`GCM`) rassemble les concepts nécessaires pour une modélisation à base de composants.

Enfin, le paquetage *Allocation modeling* (`Alloc`) permet de décrire des éléments d'association ;

2. `MARTE design model` : des concepts nécessaires pour modéliser des modèles de calculs et de communication sont introduits dans ce paquetage. En plus, des plateformes d'exécution logicielles (`SRM`) et matérielles (`HRM`) sont définies dans le paquetage *Detailed Modeling Resource* (`DRM`). Les sous paquetages `SRM` et `HRM` sont un raffinement du paquetage `GRM` ;

3. `Marte analysis model` : il contient le paquetage *Generic Quantitative Analysis Modeling* (GQAM) et ses sous paquetages *Performance Analysis Modeling* (PAM) et *Scheduling Analysis Modeling* (SAM). Il offre des analyses quantitatives de modèles, des ordonnancements et des analyses de performances ;

4. `Marte Annexes` : en annexe, Marte contient, entre autre, un langage textuel pour la spécification de valeurs (*Value Specification Language*–`VSL`). Il contient aussi des mécanismes pour la modélisation de structures répétitives (*Repetitive Structure Modeling*–`RSM`). La bibliothèque `MARTE_Library` définit des types de primitive. Ces primitives comprennent des opérations prédéfinies, couramment utilisées dans des systèmes temps réel.

Un des principaux avantages du profil Marte est la distinction claire entre les composants matériels et logiciels. Cette distinction se fait via le paquetage `DRM` contenant les stéréotypes `HwResource` et `SwResource`. Cette séparation est cruciale dans la conception de MPSoC car cela aide à partager le travail entre différentes équipes de travail.

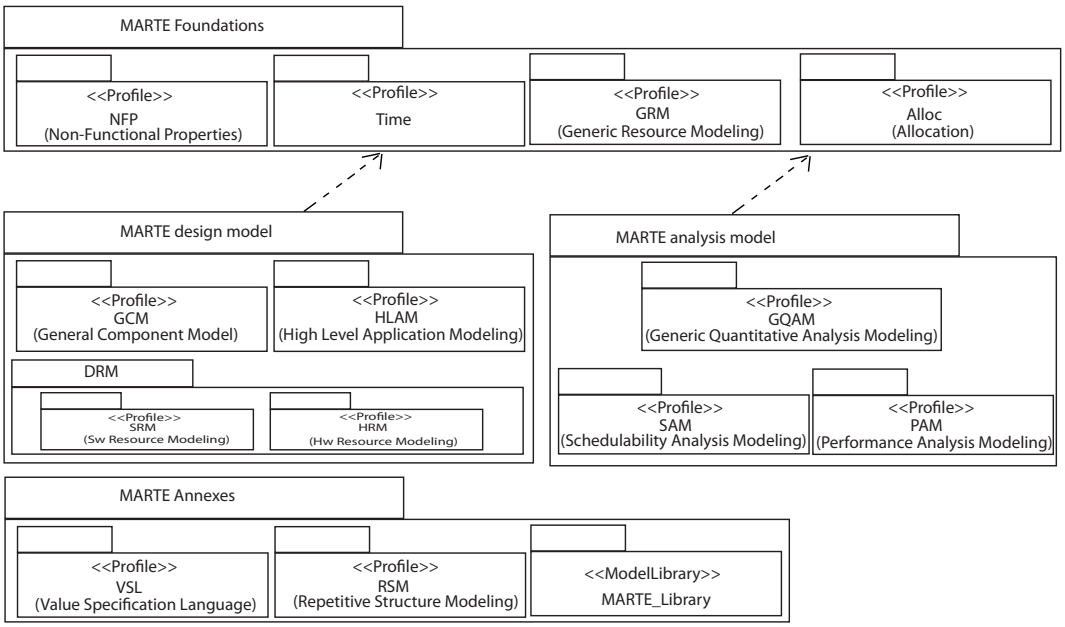


Figure 4.1: Vue globale du profil Marte.

Dans ce qui suit, nous détaillons les principaux paquetages dont les concepts sont utilisés dans cette thèse.

4.3.1.1 Modélisation des structures répétées

Le paquetage `RSM` propose un formalisme pour exprimer, d'une manière compacte, des structures répétitives. Ces structures sont décomposées en des sous composants répétitifs inter-connectés via des motifs réguliers. Dans `RSM`, des mécanismes d'accès aux différents motifs, représentant des sous tableaux, d'un tableau peuvent être utilisés pour spécifier :

- le maximum de parallélisme disponible dans une application (parallélisme de tâches et de données) ;
- des architectures massivement parallèles avec une haute précision et sous une forme compacte ;
- un placement régulier d'applications sur des architectures matérielles.

4.3.1.2 Niveau application

Des applications, modélisées dans Gaspard2, sont spécifiées à l'aide du paquetage `RSM`. Ce dernier permet de modéliser des applications manipulant des quantités importantes de données. Il adopte des représentations multidimensionnelles permettant d'exprimer tout le parallélisme potentiel présent dans des applications cibles. Les données sont structurées dans des tableaux multidimensionnels.

Deux types de parallélisme peuvent être décrits :

- *parallélisme de tâches* : il est exprimé sous la forme d'un graphe orienté de tâches acyclique. Deux tâches, n'ayant pas de dépendances de données directes entre elles, peuvent être exécutées en parallèle. Chaque lien (ou connecteur) représente une dépendance entre deux ports conformes (même type, même taille et même forme). Pour une tâche donnée, il n'y a pas de relation entre les formes des ports d'entrées et de sorties. Une tâche peut par exemple lire deux tableaux bidimensionnels et écrire un seul tableau tridimensionnel.

- *parallélisme de données* : il est exprimé à l'aide de tâches répétées. Le nombre de répétitions d'une tâche est spécifié ainsi que la manière dont les données seront consommées et produites pour chacune des répétitions. En effet, le stéréotype `shaped` de `RSM`, appliquée à une tâche fonctionnelle, indique l'espace de répétition de cette tâche (dans le jargon UML, une tâche est représentée par un composant et l'activation d'une tâche est représentée par une instance de composant). La manière dont chaque instance de composant consomme et produit des données est décrite par le concept de `Tiler`. Ce dernier contient les trois attributs `fitting`, `paving` et `origin`. Ces derniers décrivent la manière dont les tableaux d'entrées et de sorties sont respectivement consommés ou produits pour une instance de composant (ou activation d'une tâche). Une description plus détaillée des concepts sous-jacents du paquetage `RSM` est définis par Boulet et al. [22].

Modélisation des contraintes temporelles

Le paquetage TIME [9] a été introduit pour modéliser des aspects temporels de systèmes. Il enrichit le modèle de base par des informations temporelles précises mais suffisamment larges, permettant ainsi une interprétation temporelle de modèles UML.

Ce paquetage ajoute des contraintes fonctionnelles qui doivent être vérifiées ultérieurement par des techniques d'analyses et de simulations. Pour modéliser des contraintes temporelles dans Marte, nous nous servons des deux paquetages suivants :

- *TIME :* le temps peut être physique du type dense ou discret. Aussi, le temps peut être du type logique. Dans Marte, on peut définir des horloges de types logiques dont les instants dépendent de l'occurrence d'un événement ainsi que des horloges physiques dont les instants correspondent au temps physique.

  La Figure 4.2 représente la définition formelle, sous forme d'un profil, de la sémantique d'horloges dans Marte [89]. Le stéréotype `ClockType` est lié au stéréotype `Clock`. Un `ClockType` est un classificateur (`classifier`) de `Clock`, qui définit les caractéristiques du temps choisi selon ses attributs. Par exemple, l'attribut `Standard` du stéréotype `Clock` est une référence pour l'échelle de temps adoptée par l'horloge. L'attribut `nature` de `ClockType` présente deux valeurs possibles pour le temps : `discrete` ou `dense`. L'attribut `UnitType` rassemble les différents types d'unités de `ClockType` alors que l'attribut `isLogical`, du type booléen, spécifie le type de temps considéré dans `ClockType`. Quand il est à vrai, l'horloge associée référence du temps logique. Dans le cas contraire, c'est du temps physique.

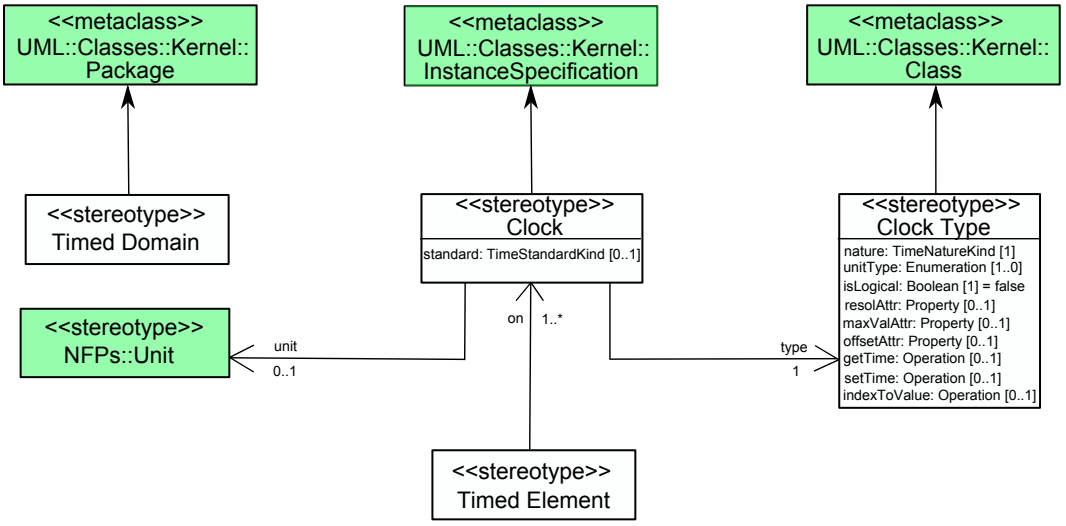


Figure 4.2: La définition du sous profil TIME de Marte [89].

- *CCSL :* le paquetage `CCSL` (*Clock Constraint Specification Language*) a été introduit de façon informelle dans le profil Marte. Il permet de spécifier dans le cadre d'UML, des contraintes aussi bien sur du temps chronométrique que sur du temps logique. Ce langage propose des mécanismes de compositions synchrones et asynchrones d'horloges logiques et ressemble en cela beaucoup au langage SIGNAL [15]. CCSL repose sur cinq principes fondamentaux de relations sur des instants :

  - *précédence stricte :* aussi noté $\prec$, est une contrainte asynchrone. Elle impose que le $k^{\text{ième}}$ instant de l'horloge de gauche précède toujours le $k^{\text{ième}}$ instant de l'horloge de droite.
  - *précédence :* aussi noté $\preceq$, est similaire à la contrainte de précédence stricte. Cependant, le $k^{\text{ième}}$ instant de l'horloge de gauche précède ou est égal au $k^{\text{ième}}$ instant de l'horloge de droite ;

- *coïncidence :* aussi noté ≡, est une forte relation synchrone. Elle impose une coïncidence paire entre les instants de chaque horloge ;
- *exclusion :* aussi noté #, dénote l'impossibilité que tous les instants d'une horloge coïncide avec les instants d'une autre horloge ;
- *périodicité :* une contrainte de périodicité, entre deux horloges clk_1 et clk_2, est vraie si les instants d'activations de clk_1 peuvent être déduits à partir des instants de clk_2 en appliquant une fonction affine. Cette fonction est caractérisée par une période et une valeur initiale (aussi connue en tant que *offset*).

4.3.1.3 Niveau architecture

Dans cette thèse nous nous intéressons plutôt aux aspects de la modélisation répétitive dans le contexte de la co-modélisation MPSoC. Par l'utilisation conjointe des paquetages HRM et RSM de Marte, nous pouvons modéliser des architectures répétitives sous une forme compacte. Cela facilitera le placement uniforme d'applications sur des architectures.

Le paquetage *Detailed Resource Modeling* (`DRM`) se décompose en deux sous paquetages *Sw Resource Modeling* (`SRM`) et *Hw Resource Modeling* (`HRM`) (le lecteur peut se référer à la Figure 4.1 pour avoir une vue globale). Le paquetage `SRM` fournit les concepts de base pour la modélisation des ressources logicielles alors que le paquetage `HRM` décrit un support de l'exécution matérielle par l'intermédiaire de différentes vues et niveaux de détail. Nous détaillons seulement le paquetage `HRM`, vu que les concepts dans `SRM` n'ont pas été utilisés dans cette thèse.

Le paquetage `HRM` décrit un support pour la description de matériels par l'intermédiaire de différentes vues et niveaux de détail. Il est composé de deux sous paquetages : `HW_Logical` et `HW_Physical`. Le paquetage `HW_Logical` offre une classification fonctionnelle des entités matérielles et des informations telles que leurs caractéristiques. Par exemple, il définit les ressources matérielles en tant qu'unités de calcul, unités de stockage et de communication (par exemple `HwBus`, `HwRam`, `HwRom`, `HwProcessor`) que nous utilisons dans cette thèse (voir Figure 4.4). Le paquetage `HwPhysical` définit une description physique de l'ensemble des ressources utilisées dans la modélisation d'une architecture. Par exemple, le sous paquetage `HW_Physical` fournit des informations supplémentaires concernant la taille, la forme, la consommation d'énergie ainsi que d'autres attributs, associés au support matériel.

4.3.1.4 Niveau association

Le profil Marte, basé sur le modèle Y, impose une séparation entre la modélisation d'applications et d'architectures. Ces deux dernières sont spécifiées séparément dans un premier lieu. Ensuite, la modélisation est suivie par une étape d'association entre une application et une architecture. L'association de composants fonctionnels (représentants de calculs) avec des composants matériels (représentants des unités d'exécution) est un aspect essentiel durant la co-modélisation de systèmes. La manière dont les composants fonctionnels et physiques sont associés a une forte influence sur les performances de systèmes et sur la consommation d'énergie totale.

Pour trouver la meilleure association, selon les critères du concepteur, toutes les associations possibles devront être testées et comparées. Cependant, quand une grande quantité de données est traitée par des architectures massivement parallèles (voir des centaines de processeurs et de co-processeurs), l'espace de solutions possibles devient

gigantesque. Il faudra donc analyser et simuler cet espace de solution avec des outils de haut niveau afin de réduire l'ensemble des solutions possibles à moindres coûts.

Le concept de placement est présent en Marte dans le paquetage `Alloc`. Nous distinguons deux types d'association :

- *simple :* une association du type *1 pour 1*, ou chaque tâche fonctionnelle est allouée à un processeur. Ce dernier est responsable de son exécution. Dans le contexte de la modélisation UML/ Marte, une association simple consiste à tracer une `dépendance` (un concept de UML) de la tâche fonctionnelle (cotée application) vers un processeur (cotée architecture). Cette dépendance est stéréotypée `allocate` (un concept de Marte). Cette opération s'effectue toujours entre deux composants ayant la même multiplicité.

- *distribuée :* ce type d'association consiste à distribuer des répétitions d'instances d'un composant fonctionnel sur des répétitions d'instances d'un composant matériel. Le principe de base de la distribution consiste à choisir un ensemble de sous répétitions d'un composant fonctionnel répété et de le placer sur une ou plusieurs répétitions d'un composant matériel. Cependant, cette opération doit être faite un certain nombre de fois jusqu'à ce que les répétitions des tâches soient au moins placées une fois sur les processeurs. En Marte, c'est avec le stéréotype `distribute` que nous spécifions cette distribution parallèle.

4.3.1.5 Niveau déploiement d'IP

Jusqu'à présent, nous nous placions à un niveau de modélisation indépendant d'une plateforme d'exécution. Par la suite, il s'agit de raffiner les modèles décrits à ce niveau vers des implémentations spécifiques selon le besoin d'un concepteur : simulation à différents niveaux d'abstraction en SystemC, synthèse d'accélérateurs matériels à l'aide de VHDL, etc. Afin de cibler chacune des implémentations possibles dans Gaspard2, le concept de propriété intellectuelle (Intellectual Property, IP) est utilisé : c'est la phase de déploiement. Un IP est un élément d'une bibliothèque, proposé sous la forme d'une boîte noir réutilisable, comme dans une approche de conception orientée composants. Ainsi dans Gaspard2, une bibliothèque d'IP est mise à disposition d'un utilisateur pour différents langages cibles : SystemC, VHDL, OpenMP Fortran, Signal, Lustre.

Les différentes implémentations d'un composant, logiciel ou matériel, sont rassemblées dans un concept que nous appelons `virtualIP`. Ces implémentations décrivent les codes sources du composant selon le langage cible. Enfin, la modélisation de haut niveau des IP logiciels et matériels, comme décrit précédemment, est concrétisée par le développement d'une libraire d'IP qu'on appelle GaspardLIB [12, 91]. Les composants matériels de SoCLib [112] ainsi que d'autres IP matériels et logiciels, développés en interne, sont intégrés dans cette bibliothèque afin d'arriver à une conception complète de MPSoC.

4.3.2 Niveaux cibles

Après avoir spécifié le système en utilisant UML et le profil Marte, le modèle résultant est alors raffiné dans Gaspard2. Ce raffinement passe par plusieurs transformations de modèles afin d'arriver à la génération de codes pour diverses technologies-cibles [50] : en langages synchrones tels que Lustre ou Signal pour la validation fonctionnelle du modèle, SystemC pour la simulation à différents niveaux d'abstraction, VHDL pour la synthèse de circuit et OpenMP pour aborder du calcul scientifique.

L'environnement Gaspard2 peut donc générer plusieurs implémentations à partir d'un système co-modélisé en Marte. La génération de code cible différentes plateformes d'exécution. Chacune des plateformes qu'on vise a des objectifs bien précis. Ces objectifs abordent l'analyse, la vérification, la simulation et le prototypage de systèmes. L'objectif final étant de concevoir correctement un MPSoC complexe en un minimum de temps et de coûts. L'environnement Gaspard2 vise la génération de différents codes à partir d'un même placement d'une application sur une architecture, à savoir :

- la génération de code VHDL correspondant à un accélérateur matériel capable d'exécuter l'application initialement modélisée à un haut niveau d'abstraction [17, 94];
- la génération de code en Lustre et Signal permet de vérifier formellement la modélisation d'une application et de détecter des problèmes tels que l'écrasement de données (les ports d'entrée et de sortie d'une tâche étant à assignation unique) [57];
- la génération de code OpenMP/Fortran rend possible l'exécution concurrente de différents processus sur une architecture multiprocesseur (architectures à mémoire partagée dans l'état actuel de Gaspard2) [117];
- la génération de code SystemC permet la simulation du comportement d'un SoC à différents niveaux d'abstraction [40].

4.3.3 Modélisation à l'aide du profil Marte

Nous présentons maintenant des exemples de modélisations de MPSoC selon la sémantique de l'environnement Gaspard2. La modélisation, reposant sur le modèle Y, divise la conception d'un système en trois étapes :

- modélisation de fonctionnalités d'une application ;
- modélisation d'une architecture multiprocesseur ;
- modélisation d'une association entre une fonctionnalité et une architecture.

Enfin, une modélisation du déploiement des IP logiciels et matériels est nécessaire afin de générer automatiquement du code vers les différentes technologies cibles.

4.3.3.1 Modélisation des fonctionnalités avec RSM

Durant la spécification UML d'une application, nous retrouvons deux concepts qui s'entrelacent fréquemment : *composant* et *instance de composant*. Un composant est une description abstraite d'un algorithme sous forme d'une boîte. Une boîte peut contenir des ports d'entrée et/ou de sortie, reflétant des données consommées et générées par l'algorithme. Une instance d'un composant représente l'activation du composant dans une mise en œuvre effective. Un composant peut contenir plusieurs instances d'un autre composant mais le contraire n'est pas vrai. Une instance de composant peut ainsi consommer ou générer des données à travers le concept de *ports*. Les composants sont classifiés en trois types :

(a) **un composant composé** : il contient plusieurs instances de composant connectées. Ces instances sont vues comme étant un graphe de tâches acycliques. Notamment, le parallélisme de tâches est vu dans ce genre de composants ;

(b) **un composant répété** : il contient une instance de composant qui est répétée un certain nombre de fois ;

(c) **un composant élémentaire** : il ne contient aucune instance de composant. Il représente une tâche atomique qui peut être vue comme étant une boîte noire. Ce type de composant est lié à une bibliothèque d'IP.

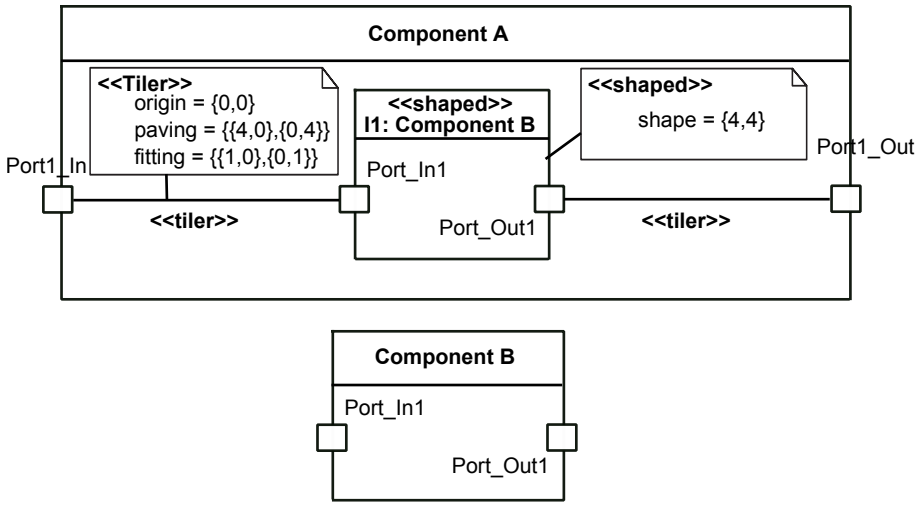


Figure 4.3: Modélisation Marte d'une application avec `RSM`.

La Figure 4.3 montre les différents stéréotypes et notations utilisés pour spécifier du calcul répétitif sur un composant fonctionnel et manipulant des structures de données multidimensionnelles. Le composant `Component A` est du type répétitif et contient une instance `I1` du composant `Component B`. Le stéréotype `Shaped`, ayant une valeur {4,4}, exprime l'espace de répétition de l'instance `I1`. Cela signifie que le composant est répété (4×4) fois. Chacune des répétitions consomme des motifs en entrées. Ces motifs sont extraits à partir des tableaux d'entrées du composant englobant `Component A`. De la même manière, chacune des répétitions produit des motifs en sortie. Ces derniers seront consommés par des tableaux en sortie du composant `Component A`.

Pour décrire la répétition d'un composant, l'espace de répétition doit être spécifié ainsi que les attributs `fitting`, `paving` et `origin` associés au stéréotype `tiler`. L'attribut `fitting`, du type matrice, permet de déterminer les éléments d'un tableau qui sont liés à chaque motif. La matrice `paving` décrit l'espacement régulier des différents motifs au sein d'un tableau d'entrée ou de sortie. En d'autres termes, elle permet d'identifier l'origine de chaque motif associé à chaque répétition d'une instance de composant. Le vecteur `origin` spécifie l'origine du motif initial dans un tableau.

La description du parallélisme de tâches et de données offerte par le paquetage `RSM` permet une distribution efficace de tâches fonctionnelles sur une architecture multiprocesseur.

4.3.3.2 Modélisation de la plateforme d'architecture

Nous montrons maintenant un exemple d'une architecture multiprocesseur, à mémoire partagée. Cette architecture, modélisée en Marte est illustrée dans la Figure 4.4. L'architecture modélisée contient quatre instances `P1`, `P2`, `P3` et `P4` d'un composant `Processor1`. Ces instances représentent quatre processeurs distincts ayant les mêmes

spécifications que `Processor1`. Ces processeurs communiquent entre eux à travers une mémoire partagée. Nous définissons également deux instances de mémoires `I1` et `d1` de types respectivement `hwRam` et `hwRom` pour écrire, stocker et récupérer respectivement des données et des instructions. L'accès aux mémoires se fait par l'intermédiaire d'un bus partagé localisé entre les processeurs et les deux mémoires partagées. Toutes les instances de composants, contenues dans le composant `Architecture`, sont stéréotypées avec des concepts du profil Marte (tous les noms commençant par `hw`). Toutes les communications entre les différents processeurs passent forcément par l'instance `B` du composant `Bus`, stéréotypé `HwBus`. Les données échangées entre les différents processeurs et les mémoires, à travers le bus, passent à travers les ports `Master` et `Slave` de l'instance `B`. Un émetteur (l'instance `Cam` du composant `Sensor`), stéréotypé `hwSensor`, et un récepteur (l'instance `A1` du composant `Actuator`), stéréotypé `hwActuator`, sont dédiés à la production et la consommation de données.

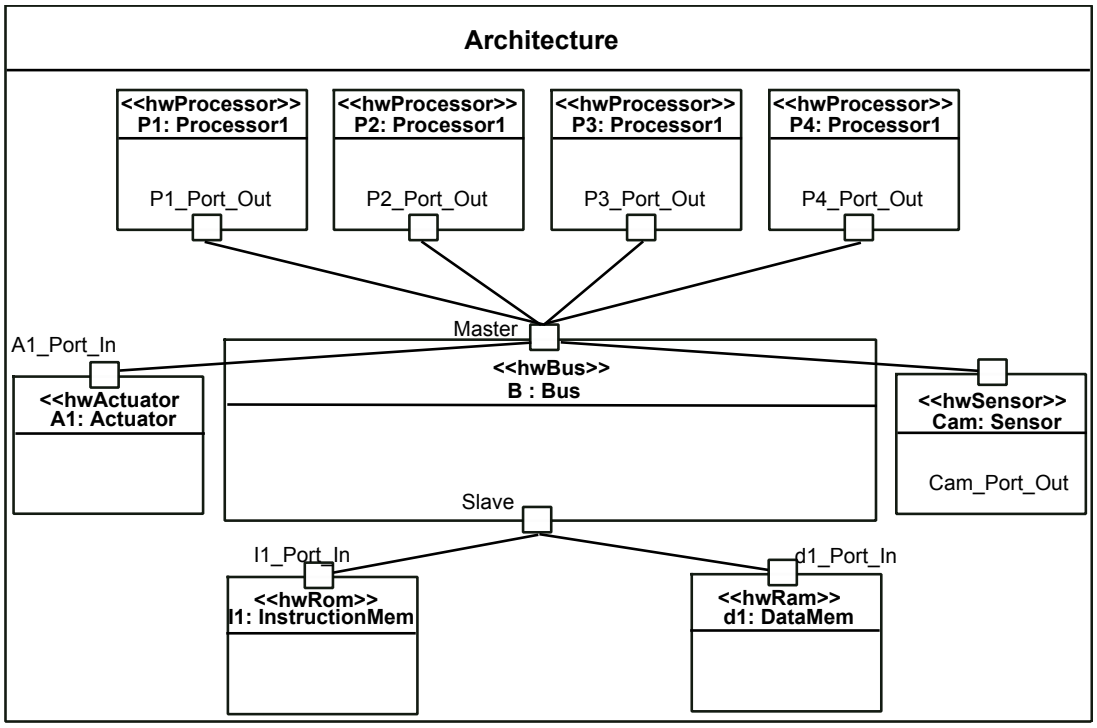


Figure 4.4: Modélisation Marte de l'architecture considérée.

Le concept de stéréotype, évoqué précédemment, enrichit le modèle UML initial avec des propriétés concernant les systèmes embarqués temps réel, typiquement la fréquence d'un processeur ou la taille et le type d'une mémoire.

4.3.3.3 Modélisation de l'association

Dans le monde du co-design, une association permet d'allouer un ensemble de composants fonctionnels sur des ressources physiques. Dans le cas de Gaspard2, nous allouons des tâches fonctionnelles à des processeurs et des ports d'entrée/sortie à des mémoires.

L'étape d'association joue un rôle très important sur les performances globales du système, d'où la nécessité de guider l'utilisateur à choisir les meilleurs choix d'associations. En effet, un grand nombre d'associations est possible pour une application et une architecture données. Le concepteur ne doit choisir que la ou les configurations les plus

adaptées à ces critères (nombre maximal de processeurs, taille de la puce, quantité de mémoire disponible, consommation d'énergie, temps d'échéance, etc.).

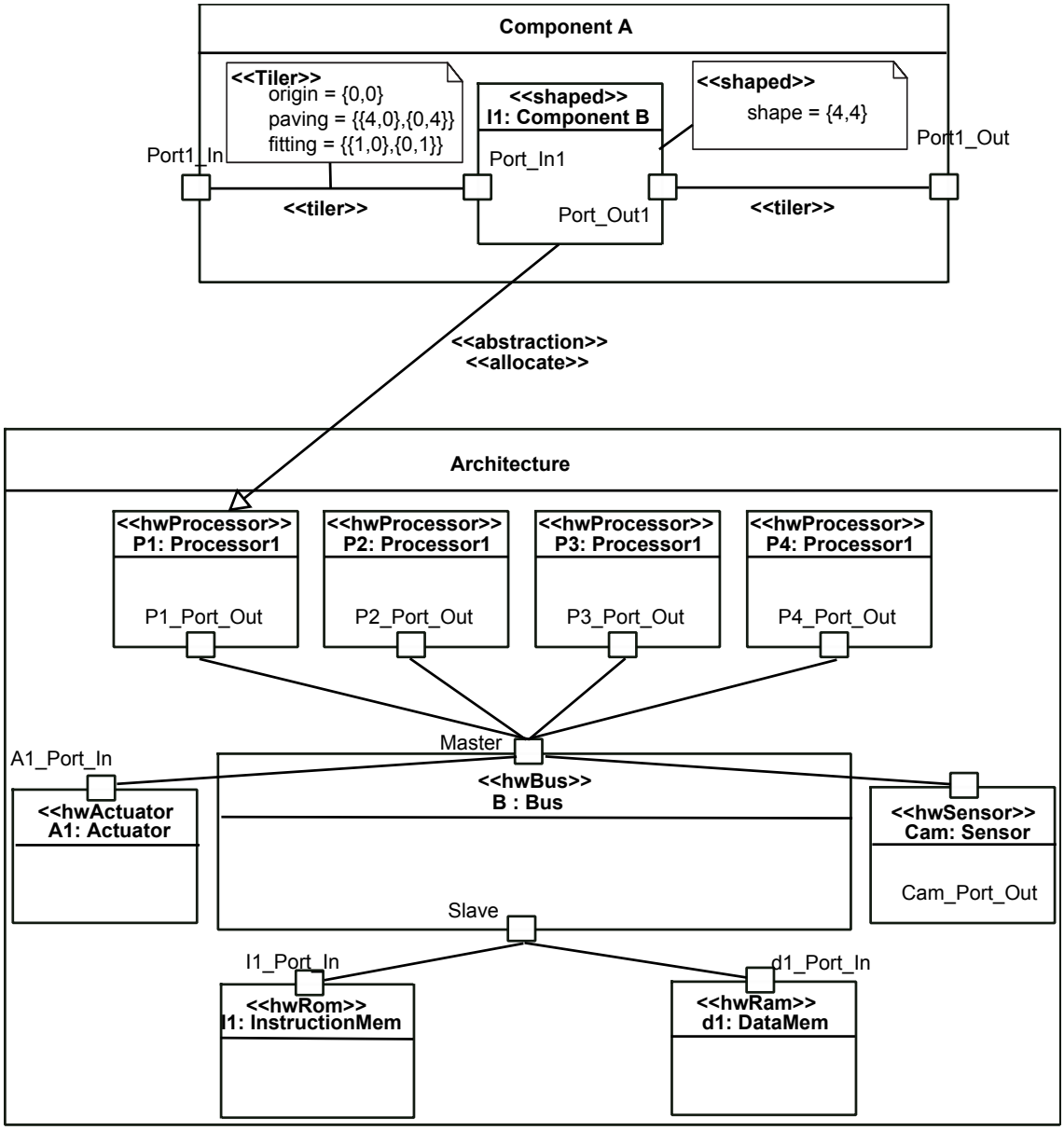


Figure 4.5: Modélisation Marte de l'association entre les fonctionnalités et une architecture choisie.

Dans la Figure 4.5, l'instance `I1` du composant fonctionnelle `Component B`, contenue dans le composant `Component A`, est associée à l'instance `P1` du processeur `Processor1`. Cette association est faite par le biais d'un connecteur UML stéréotypé `allocate` de Marte. Cela signifie que `Processor1` va prendre en charge l'exécution des quatre répétitions de l'instance `I1`.

4.3.3.4 Modélisation du déploiement

Jusqu'à présent, nous nous placions à un niveau de modélisation indépendant d'une plateforme d'exécution. Durant la phase de déploiement dans Gaspard2, on intro-

duit la notion de propriété intellectuelle (Intellectual Property, IP) qui permet de cibler différentes implémentations spécifiques. Nous passons ainsi à une modélisation dépendante de la plateforme d'exécution. Chaque ressource est associée à une implémentation matérielle. Le type et les caractéristiques des ressources physiques sont aussi définis. La Figure 4.6 montre comment un processeur matériel `Processor1` est déployé sur `VirtualProcessor1`. Ce dernier, du type `virtualIP`, contient deux implémentations matérielles au niveau *SystemC Transaction Level Modeling* (TLM). Le composant `VirtualProcessor1-TLM1` (resp. `VirtualProcessor1-TLM2`), stéréotypé `hardwareIP` et `hardwareProcessor`, a une fréquence de 100 MHz (resp. 200 MHz). Nous avons choisi le processeur ayant comme fréquence 100 MHz. Ce choix est présenté dans la Figure 4.6 par la flèche stéréotypée `implements` allant de `VProcessor1-TLM1` vers `Processor1`.

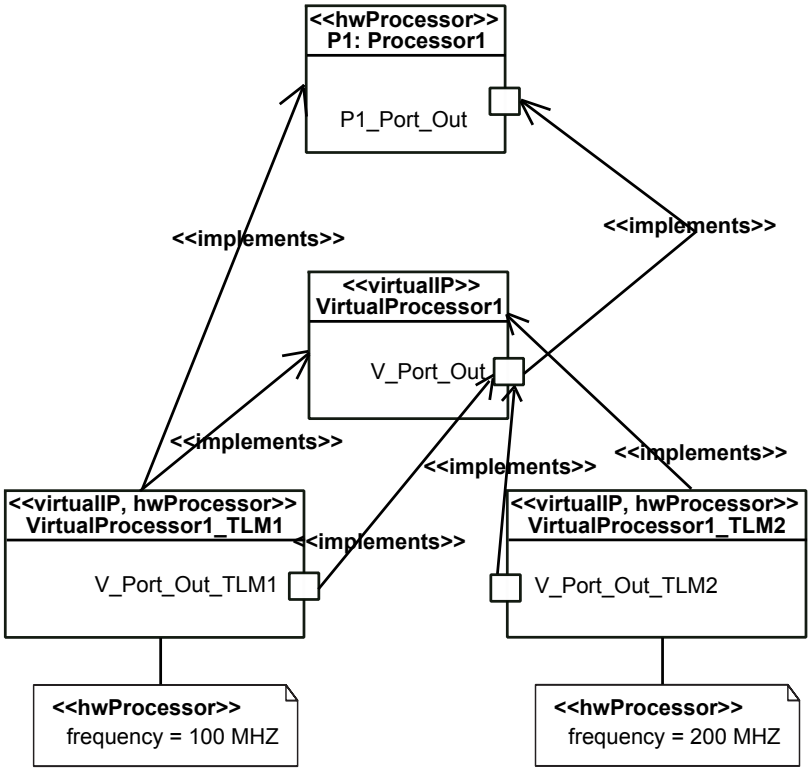


Figure 4.6: Modélisation Marte du déploiement matériel.

4.3.4 Génération de code par transformations de modèles

Parmi les diverses activités de recherche consacrées aux problèmes de conception de logiciels, l'ingénierie dirigée par des modèles (IDM) [90] se présente comme une des approches les plus prometteuses liées à la conception et le développement des systèmes embarqués temps réel. C'est une approche de développement de logiciels dont la brique de base est liée aux modèles.

L'IDM vise principalement à rendre la notion de modèle l'élément essentiel dans le domaine du génie logiciel. Le but est d'avoir un environnement de développement de logiciel qui permet le passage d'un modèle à un autre. Au lieu de construire et de reconstruire des systèmes après avoir changé une partie d'une application donnée, des

modèles sont étendus, sélectionnés ou échangés avec d'autres modèles permettant ainsi de recréer le système de nouveau sans aucun coût ou effort considérable.

Modèle et métamodèle

Pour qu'un modèle soit défini, une description formelle prédéfinie de ce dernier doit être faite. C'est le rôle d'un métamodèle ou du *modèle de modèle*. Un métamodèle contient un ensemble de concepts (entité, relations, conditions, etc.) permettant ensemble d'abstraire un certain domaine. En d'autres termes, un modèle décrit un phénomène à un certain niveau d'abstraction alors que le métamodèle de ce modèle décrit les propriétés et les contraintes imposées sur le modèle lui-même. Comme exemple concret, un code source Java contenant des erreurs syntaxiques ne sera jamais compilé. Le code source de ce programme représente le modèle alors que le compilateur qui vérifie le code représente le métamodèle. La description des modèles et des métamodèles est codifiée en utilisant un langage de description de modèle tel que UML [85] ou EMF [41].

Transformations de modèles

Les modèles que nous manipulons dans l'IDM sont spécifiés à différents niveaux d'abstraction et parfois dans des langages différents. Dans le développement de modèle, le défi majeur est de transformer des modèles de haut niveau indépendants de la plateforme d'exécution (par exemple le niveau algorithmique) vers des modèles spécifiques à une plateforme (par exemple au niveau RTL), pour finalement générer des modèles de très bas niveau (voir niveau implémentation ou physique). La transformation de modèle dans l'IDM consiste à transformer un ou plusieurs modèles sources en un ou plusieurs modèles cibles, chaque modèle étant conforme à un métamodèle [18, 116].

4.4 Analyse et vérification dans Gaspard2

Nous avons montré comment modéliser un MPSoC en Gaspard2. La modélisation commence par une spécification d'une application et d'une architecture multiprocesseur. Ensuite, une association est modélisée entre des composants fonctionnels et des ressources physiques. Cette étape est suivie par un déploiement d'IP logiciels et matériels.

Cependant, durant les différentes étapes de modélisation, plusieurs choix de configurations sont possibles :

- durant la modélisation d'une application, le découpage fonctionnel des différentes tâches peut être fait de plusieurs manières. Par exemple, pour une même distribution de tâches, nous pouvons modifier la taille des données manipulées selon la hiérarchie de composants ;
- dans une architecture multiprocesseur, un grand nombre de processeurs peut être défini. Cependant, il est utile de savoir le nombre exact de processeurs adaptés à l'exécution des fonctionnalités ;
- plusieurs configurations d'associations sont possibles entre une application et une architecture.

Nous avons vu que, pour concevoir un système, plusieurs configurations sont possibles. Pour cela, des méthodes de vérification, de simulation et de prototypage sont définies afin de trouver la configuration idéale du système.

4.4.1 Vérification formelle

La vérification fonctionnelle est l'une des phases qui prend le plus de temps parmi les différentes phases de conceptions de systèmes. Le processus de vérification, une étape cruciale dans le processus de conception de SoC, est un travail qui implique parfois plusieurs équipes d'ingénieurs. Ces équipes coopèrent ensemble afin de concevoir au final un système complet sans faille. Des tests rigoureux et des processus de vérification peuvent révéler des problèmes avant d'arriver à la fabrication du produit final. Si un produit mal conçu est construit, cela entraîne un travail très coûteux pour régler le problème. Si des problèmes sont non détectés durant la phase de vérification et suffisamment importants, ils peuvent empêcher le lancement du produit sur le marché au bon moment.

Les applications nécessitant du calcul intensif peuvent être spécifiées à un très haut niveau d'abstraction en utilisant l'environnement Gaspard2. Cependant, le concepteur de système peut commettre des erreurs de spécifications. Ces erreurs peuvent être fondamentales et nocives à un système. C'est pourquoi la vérification de fonctionnalités, d'un système spécifié en Gaspard2, est très importante et cruciale. Il existe actuellement quelques outils qui permettent la vérification, à faibles coûts, de certaines propriétés attendues au niveau application. Nous les classifions en deux parties :

- les incohérences reliées à la bonne utilisation des notations UML dans Gaspard2. Ce genre de problème peut être vérifié par l'insertion des contraintes OCL, les outils UML graphiques, par les transformations de modèles, etc.

- les problèmes concernant la violation de la sémantique de Gaspard2 comme par exemple l'assignation unique des éléments de tableaux et les dépendances de données acycliques. Ce genre de problème a été traité par Yu [57, 136] durant ces travaux de thèse. La proposition de solution est de transformer le graphe de dépendance de données en une annotation synchrone en Lustre et Signal. Cette transformation aboutit à la génération automatique, à partir d'un modèle d'application, de code synchrone en Lustre et Signal. Les outils synchrones permettent ensuite de vérifier automatiquement ce genre de contraintes en compilant le code synchrone généré.

4.4.2 Simulation, exécution et synthèse à l'aide de SystemC, OpenMP et VHDL

Un flot de conception systématique de MPSoC est défini dans Gaspard2. Ce flot est dédié au développement d'applications hautes performances exécutées sur des architectures massivement parallèles. À travers une génération automatique de code source pour plusieurs technologies parmi lesquelles SystemC, OpenMP et VHDL, Gaspard2 permet la simulation, l'exécution et le prototypage de modèles UML/ Marte. Une approche Ingénierie Dirigée par des Modèles (IDM) est définie pour la mise en œuvre de ce flot de conception. Le code généré des trois technologies cibles servira à :

- *la simulation* : un environnement de co-simulation logicielle/matérielle de MPSoC a été proposé par Benatitallah durant ces travaux de thèse [12]. Il décrit un modèle UML/ Marte en SystemC au niveau transactionnel (TLM). Cet environnement contient des modèles d'estimation de performance afin de rendre possible l'évaluation du temps d'exécution d'une application. Ce paramètre constitue un premier critère de sélection entre les différentes solutions architecturales possibles. Il intègre aussi

des outils flexibles pour l'évaluation de la consommation d'énergie à l'aide de modèles de consommation d'énergie à différents niveaux d'abstraction. Cette métrique constitue un deuxième critère de sélection des solutions architecturales dans une exploration de l'espace de solutions ;

- *l'exécution :* Une méthodologie d'IDM, proposé par Taillard [117], permet une exécution parallèle de modèles spécifiés en UML/ Marte. Les cibles de cette méthodologie sont des machines à mémoires partagées (classe dont font partie les machines ayant des processeurs multi-cœurs) programmées avec le standard OpenMP Fortran, le code source étant généré automatiquement à travers des techniques de transformations de modèles ;

- *la synthèse de circuit :* un flot de conception est développé permettant la transformation d'une application modélisée à haut niveau d'abstraction (UML) vers un modèle RTL. Ce modèle est conforme à un métamodèle décrivant des concepts utilisés au niveau RTL. Par ailleurs, ce métamodèle considère la technologie d'implémentation FPGA (*Field-Programmable Gate Array*) et propose différents niveaux d'abstractions pour une même FPGA. Ces multiples niveaux d'abstractions permettent un raffinement des implémentations matérielles. En fonction des contraintes de surfaces disponibles (technologie FPGA), le flot de conception optimise le déroulement des boucles et le placement des tâches. Le code VHDL, généré automatiquement, peut être directement simulé et synthétisé sur un FPGA [17, 94].

4.4.3 Exploration et optimisation du domaine de conception

Plusieurs travaux ont été faites, dans le cadre du projet Gaspard2, afin d'explorer l'espace de conception de systèmes co-modélisés selon le modèle Y :

- `Corvino et al.` [32] proposent une méthode de configurations de modèles architecturale pour ASICs, adaptées à l'exécution d'applications hautes performances. Les applications et les architectures, contenants des contraintes de configurations, sont basées sur le modèle de calcul Arrray-OL. Les architectures considérées se composent de processeurs parallèles possédant chacun une mémoire locale. Cette dernière permet la sauvegarde de données transférées entre des processeurs dans n'importe quel ordre. Une méthode d'exploration du domaine de conception permet de choisir les configurations d'associations qui améliorent le temps d'exécution et qui diminuent les tailles de mémoires tampons ;

- `Redjedal et al.` [8] définissent un algorithme hybride consistant d'une méthode évolutionniste et d'une heuristique. La méthode évolutionniste permet de retrouver la configuration d'association de tâches sur une architecture multiprocesseur. Les auteurs visent une optimisation bi-objective (temps et énergie) du placement d'un graphe hiérarchique d'application sur un graphe hiérarchique d'architecture. L'autre partie de l'algorithme, basée sur une heuristique, permet l'optimisation de communications ;

- `MEENA Ashish` [78] proposent dans sa thèse une optimisation multi-objectives pour trouver le nombre de ressources physiques à utilisé jusqu'à un certain goulot d'étranglement qui limiterait leur utilisation (exploration de matériel). Après ce calcul des ressources utiles, une heuristique exploite les systèmes à plein, de l'utilisation des ressources à celle de la bande passante.

4.5 Vers une approche d'exploration de l'espace de conception

Dans ce chapitre, nous avons présenté Gaspard2, un environnement de co-modélisation de MPSoC avec UML/ Marte. La modélisation de MPSoC, avec un langage de haut niveau, facilite l'analyse, la vérification et la validation de systèmes complexes sur différentes technologies. Étant basé sur le profil UML/ Marte, un modèle Gaspard2 est facilement compréhensible par différents concepteurs. De même, l'utilisation de la programmation à base de composants et l'intégration d'une partie de déploiement d'IP facilite la réutilisation d'anciens modèles. Cela contribue énormément à réduire les coûts de conceptions.

Cependant, Gaspard2 manque de techniques d'analyse et de vérification, très tôt dans le flot de conception, de l'espace de solutions possibles de MPSoC. En effet, la conception d'une application haute performance (contenant une grande quantité de données) sur une architecture multiprocesseur (massivement parallèle) comportent un grand espace de solutions possibles. Une solution représente une configuration d'une application, d'une architecture et de l'association entre les deux dernières. Notre contribution repose sur l'analyse et la vérification de l'ensemble des solutions possibles, très tôt dans le flot de conception. Le but final est de réduire cet espace, en nombre limité de configurations possibles.

Nous présenterons dans la partie suivante, et dans le cadre de l'environnement Gaspard2, une méthodologie d'analyse et de vérification de MPSoC à base d'horloges abstraites. Un système, modélisé conjointement en Marte selon le modèle Y, est décrit par le biais d'horloges abstraites. Nous utilisons ces horloges comme support d'analyse et de vérification.

Afin de respecter la conception selon le modèle Y, nous distinguons trois types d'horloges abstraites :

- `horloges fonctionnelles` : ces horloges ont pour but de capturer la quantité ou la charge de travail dans une application. Elles conservent aussi les dépendances de données imposées soit par construction soit implicitement par le concepteur ;

- `horloges physiques` : ce type d'horloges abstraites traduit les vitesses des différents processeurs sous formes d'instants logiques (chaque processeur étant alloué une horloge). Les instants des différentes horloges sont synchronisés avec des instants d'une horloge fictive qu'on appelle horloge de référence ;

- `horloges d'exécutions` : nous proposons une projection des instants logiques d'horloges fonctionnelles sur les instants des horloges physiques associées. Cette projection se base sur un algorithme d'ordonnancement statique non préemptif. Elles ont pour but de simuler l'activité des différents processeurs durant l'exécution de fonctionnalités.

Nous nous basons sur ces trois horloges pour vérifier des contraintes fonctionnelles et non fonctionnelles d'un modèle co-modélisé en Marte. Nous abordons dans notre analyse les propriétés suivantes :

- **fonctionnelles** : durant la modélisation de fonctionnalité en Marte, des contraintes de pécédences et de périodicités sont présentes. Ces contraintes sont imposées par la sémantique du modèle de calcul sous-jacent du paquetage RSM. Ces contraintes représentent entre autres l'ordre d'exécution des différents composants élémentaires. Le concepteur peut aussi ajouter manuellement, par le biais des

horloges synchrones des paquetages TIME et CCSL, des contraintes de périodicité ou de précédences. Ces contraintes additionnelles enrichissent la modélisation du comportement fonctionnelle (en ajoutant par exemple des contraintes de qualité de service) ;

- **non fonctionnelles** : pour une fonctionnalité donnée, un grand espace de configuration possible d'architecture et d'association existe. La simulation de toutes les solutions, au niveau TLM par exemple (ou pire au niveau RTL), prend beaucoup de temps. Pour cela, notre objectif est de restreindre les configurations possibles en analysant les trois facteurs suivants :
  - le nombre de processeurs impliqués dans l'exécution : cette valeur peut aller d'un processeur jusqu'au nombre maximal de processeurs disponibles ;
  - les fréquences minimales des processeurs : la fréquence minimale d'un processeur est celle qui permet la terminaison de l'exécution d'une tâche sans dépasser le temps d'échéance ;
  - différents type d'association : pour une application et une architecture données, un grand nombre d'association est possible. Il faut cependant choisir celle qui est la plus convenable selon les critères du concepteur.

  Ces trois facteurs affectent énormément l'exécution globale du système. La modification d'au moins un des trois facteurs affectera directement la consommation d'énergie totale du système, le temps d'exécution global et le respect des contraintes de périodicités et de précédences. Donc pour chaque modification, une analyse et une vérification des différentes contraintes fonctionnelles et non fonctionnelles doivent être appliquées.

Part II

Conception et analyse de MPSoC basées sur Marte et des horloges abstraites

Présentation générale

Dans cette partie, nous nous intéressons à l'abstraction de modèles Gaspard2 avec des notions d'horloges abstraites. La conception de MPSoC, basée sur le modèle Y, suit l'approche de conception *top-down*. À l'aide des horloges abstraites, nous présentons une méthodologie d'analyse temporelle à un haut niveau d'abstraction. Cette méthodologie, respectant elle aussi le modèle Y, fait une séparation claire entre une application, une architecture et l'association des deux dernières. Notre objectif est de trouver, très tôt dans le flot de conception, les meilleures configurations d'architectures et d'associations entre une application et une architecture. Cela permet de réduire l'espace de solutions possibles de configurations de systèmes.

L'approche générale est illustrée par la Figure 4.7. Au début, nous modélisons une application, suivie par une analyse de synchronisation de composants fonctionnels. Ensuite, nous proposons plusieurs configurations possibles d'architectures en analysant la vitesse des différents processeurs. Finalement, la conception se conclut par une étape d'association de tâches fonctionnelles sur des ressources physiques. À partir de cette étape, nous proposons une abstraction de cette association par le biais d'horloges abstraites d'exécutions. Ces horloges simulent l'activité des différents processeurs durant l'exécution de fonctionnalités. Nous reposons sur ces horloges pour estimer les performances d'un système, pour vérifier des contraintes de périodicité et de précédence et enfin pour estimer la consommation d'énergie totale d'un système.

Les horloges abstraites, décrivant le comportement temporel d'un système, permettent de décrire :

1. une application en marquant les activations de tâches fonctionnelles et leurs dépendances de données ;
2. une architecture en marquant la vitesse de fonctionnement des processeurs ;
3. une association en traçant l'exécution des fonctionnalités par des processeurs.

Le but final est d'aider le concepteur à faire des choix de configurations qui lui seront convenables et qui répondent aux critères suivants : vérification de contraintes de précédence et de périodicité, respect des temps d'échéances imposés sur une application et minimisation de la consommation d'énergie globale. Une grande partie de ces travaux a été publiée dans [5, 3, 6, 4, 2].

Les chapitres 5, 6 et 7 sont consacrés à la description de notre apport personnel et notre contribution à la résolution de problèmes d'analyse et de vérification rapide d'applications hautes performances implantées sur MPSoC.

Dans le chapitre 5, nous montrons comment modéliser en Marte des applications hautes performances. Ensuite, nous proposons une abstraction de ces fonctionnalités par le biais d'horloges abstraites fonctionnelles. Ces horloges, de types binaires, nous servirons pour analyser formellement des contraintes fonctionnelles et non fonctionnelles à l'aide des outils synchrones.

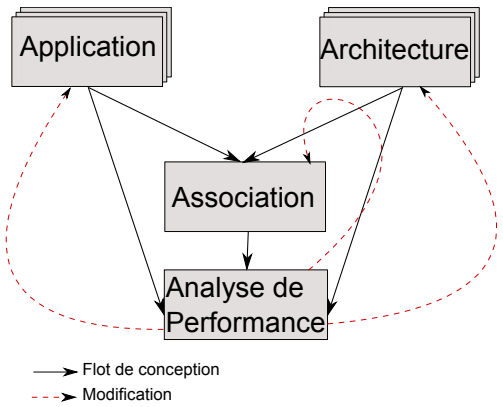


Figure 4.7: Spécification et analyse de MPSoC basées sur le modèle Y.

Dans le chapitre 6, nous montrons comment modéliser en Marte des architectures multiprocesseurs. Ensuite, nous proposons une analyse de cadencement des processeurs à l'aide d'horloges abstraites physiques. Dans une étape suivante, nous modélisons en Marte une association entre une application et une architecture. Nous proposons ensuite une projection d'horloges fonctionnelles sur des horloges physiques. Le résultat de cette projection est une nouvelle trace d'horloges abstraites, simulant l'activité des différents processeurs durant l'exécution de fonctionnalités.

Dans le chapitre 7, nous proposons une méthodologie d'analyse et de vérification de contraintes fonctionnelles et non fonctionnelles. Cette analyse repose sur des traces d'horloges abstraites d'exécutions obtenues dans le chapitre 6. Nous essayons par cette analyse de réduire l'espace de solutions possibles de configurations d'un système co-modélisé en Marte. Les différentes analyses offrent au concepteur suffisamment d'informations pour être capable de configurer correctement un système, très tôt dans le flot de conception.

Chapter 5

Modélisation de comportements fonctionnels contraints

5.1 Introduction

L'état de l'art, exposé dans les chapitres 2 et 3, a montré que certains outils d'analyse et d'exploration de MPSoC existants, abordent les différentes problématiques exposées dans le chapitre 1. C'est bien le cas de l'environnement `NESSI`. Nous partageons avec cet environnement le haut niveau d'abstraction dans la conception et l'analyse de systèmes. Nous partageons aussi la séparation de vues entre la conception d'applications, d'architectures et de l'association entre de fonctionnalités et de ressources physiques. Nous suivons en cela la conception selon le modèle Y. Cependant, dans le cadre de cette thèse, nous manipulons des applications hautes performances, contenant une grande quantité de données. Nous considérons aussi des architectures massivement parallèles. Cela rend l'analyse très fastidieuse et complexe. Nous pensons que notre travail apporte une contribution dans l'analyse de tels systèmes, à un très haut niveau d'abstraction.

Dans ce chapitre, nous nous intéressons à l'abstraction de comportements fonctionnels de MPSoC. Le point de départ est un modèle Marte, spécifié à l'aide du *diagramme de structure composite* de UML, décrivant des fonctionnalités. Ce modèle contient plusieurs niveaux de parallélisme potentiel. Pour cela, nous proposons de raffiner le modèle de calcul sous-jacent (reposant sur le paquetage RSM) en choisissant un des trois niveaux de

parallélisme suivants : `parallèle`, `pipeline` ou `mixte`. Cela aboutit à une déduction de contraintes de périodicité et de précédences du modèle fonctionnel. Le concepteur peut manuellement imposé ce raffinement en se servant des concepts d'horloges synchrones définis dans les paquetages TIME [9] et CCSL [74] du profil Marte.

La section 5.2 résume les principes de la modélisation de fonctionnalités en Marte, notamment la présentation des sémantiques du langage pour l'expression du parallélisme de tâches et de données. Les règles de la définition du raffinement de modèles de calcul sont décrites dans la section 5.3. Dans la section 5.4, nous définissons d'abord la sémantique des horloges abstraites fonctionnelles. Ensuite nous montrons comment spécifier explicitement des contraintes de dépendances de tâches dans un modèle Marte.

5.2 Spécification d'algorithmes à l'aide de Marte

Nous présentons dans cette section la sémantique de la modélisation de fonctionnalités en Marte. D'abord nous introduisant la sémantique d'un graphe de tâches fonctionnelles. Ensuite nous présentons des contraintes de dépendances de tâches qui sont imposées par constructions de modèles.

5.2.1 Graphe de dépendances de tâche

La modélisation de fonctionnalités dans Marte repose fortement sur le paquetage RSM. Ce dernier est inspiré du modèle de calcul *Array Oriented Language* (ArrayOL) [37]. Il permet de modéliser des applications manipulant de grandes quantités de données.

Dans un graphe de tâches, des tableaux (ou motifs), de tailles constantes, sont produits et consommés par une tâche. Différentes tâches peuvent être reliées entre-elles par des dépendances. Lorsqu'une dépendance est spécifiée entre deux tâches, cela signifie que l'une d'entre elles requiert des données de la part de l'autre afin de s'exécuter. Cette dépendance représente une communication producteur-consommateur. Cela exige un ordre partiel minimal d'exécution.

À partir de ce constat, nous pouvons déduire des contraintes d'activations de tâches dans le graphe global de tâches. Avec activation, nous entendons l'exécution d'une fonctionnalité comportant la lecture, le traitement et l'écriture de données.

Nous adoptons en cela la théorie synchrone [14][1] décrite formellement par Potop-Butucaru et al. [93].

Les applications peuvent être composées hiérarchiquement et à plusieurs niveaux de spécifications :

- Au plus haut niveau de spécification se trouve un composant du type composé. Ce dernier contient différentes autres instances de composants, qui à leurs tours, sont de types composés ou répétitifs ;

- Au plus bas niveau dans la spécification se trouve des composants de types élémentaires. Ces composants sont vus comme des boîtes noires. Des IP logiciels sont déployés ultérieurement dans une étape de déploiement d'IP.

[1] L'hypothèse synchrone considère les machines programmées comme étant infiniment rapide et capable de respecter les contraintes imposées par l'environnement. Les calculs et les communications ont un temps nul.

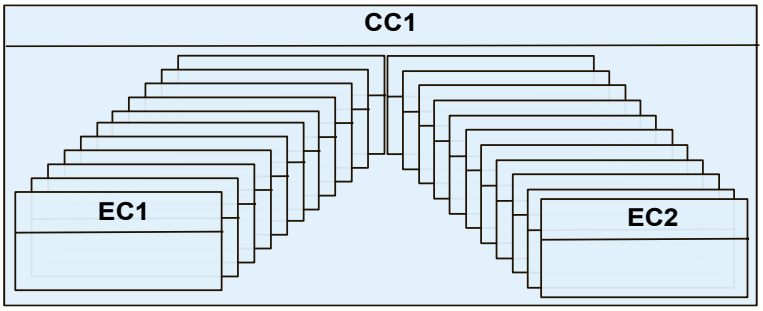


Figure 5.1: Description informelle des niveaux de spécification dans un graphe de tâches fonctionnelles.

Un exemple informel de cette hiérarchie de composants est montré dans la Figure 5.1. Au plus haut niveau, le composant composé `CC1` englobe l'ensemble des instances de composants du graphe hiérarchique. Au plus bas niveau sont `EC1` et `EC2` représentant des instances de composants élémentaires. Entre le premier et le dernier niveau de spécification, des composants de types composés ou répétitifs peuvent être imbriqués.

5.2.2 Contraintes de dépendance de tâches

Dans un modèle Marte, la description graphique de tâches permet d'exprimer les accès aux données et en même temps les dépendances de données, donc l'ordre d'exécution des différentes tâches élémentaires. Chaque tâche élémentaire est exécutée un certain nombre de fois selon son espace de répétition total. Cet espace de répétition peut être calculé en parcourant l'espace de répétition de tous les niveaux intermédiaires dans la hiérarchie de tâches à partir du niveau le plus haut jusqu'au niveau le plus bas.

Dans le cadre de l'analyse d'horloges abstraites que nous proposons dans cette thèse, les contraintes de dépendance de tâches, déduites du modèle d'application, sont associées uniquement aux composants de types élémentaires. Enfin, seuls les composants élémentaires peuvent être associés à du calcul, donc seuls ces composants sont exécutés. Ainsi, en remontant dans le niveau de spécification d'un composant élémentaire (au plus bas niveau) vers le premier composant du type composé (au plus haut niveau), nous calculons l'espace de répétition total du composant élémentaire selon l'espace de répétition de chaque niveau supérieur et nous déduisons l'ordre d'exécution des différents composants (ou tâches) élémentaires.

5.3 Raffinement préliminaire vers des modèles d'exécution

La description de fonctionnalités en Marte est représentée par un graphe acyclique de tâches fonctionnelles. Ce graphe exprime un haut niveau de parallélisme de tâches et de données. Nous proposons dans cette section de raffiner ce niveau de parallélisme en considérant trois modèles d'exécution : `parallèle`, `pipeline` et `mixte`.

5.3.1 Modèle `parallèle`

Dans un modèle d'exécution `parallèle`, nous exploitons au maximum le niveau de parallélisme existant dans une spécification fonctionnelle. Cela est fait en allouant pour toutes les instances de composants élémentaires des processeurs virtuels. Le choix

d'utiliser particulièrement ce type d'exécution permet de profiter au maximum du haut niveau de parallélisme de tâches sous-jacent dans un modèle Marte.

Les contraintes qui sont imposées sur ce type d'exécution sont les suivantes :

1. toutes les répétitions d'un composant élémentaire sont exécutées en parallèle ;
2. deux composants élémentaires qui n'ont pas une dépendance directe entre eux sont exécutés en parallèle ;
3. quand deux composants élémentaires sont dépendants l'un de l'autre à travers des connecteurs, le composant producteur de données est exécuté avant ou au même moment que le composant consommateur.

La Figure 5.2 montre un exemple d'une application modélisée en Marte. Le composant `CC_Main`, du type composé, est le premier composant dans le niveau de hiérarchie. Il englobe les instances de composants `IC1`, `IC2`, `IC3` et `IC4`. Pour des raisons de lisibilité, nous modélisons seulement la hiérarchie des instances `IC1` et `IC2`. L'instance `IC1`, par exemple, représente le composant `RC1` qui est du type répétitif. Le composant `RC1` englobe à son tour l'instance de composant `I1` avec un espace de répétition multidimensionnel {3,3}, représenté par le stéréotype `shaped` (au centre de la Figure 5.2). Cela implique que le composant élémentaire `EC1` de l'instance `I1` va être instancié et exécuté neuf fois. Quand le modèle d'exécution `parallèle` est choisi, ces instances de composants sont allouées chacune à un processeur virtuel et peuvent être exécutées simultanément (contrainte 1). Pour les composants élémentaires `EC1` et `EC2`, il existe en total 21 processeurs virtuels vu qu'il y a $3 \times 3 = 9$ instances du composant `EC1` et $4 \times 3 = 12$ instances du composant `EC2`.

Des contraintes de dépendances de tâches sont imposées, par construction, sur le graphe de tâches global selon la sémantique de RSM. Ces contraintes imposent un ordre d'exécution des différentes tâches fonctionnelles représentées par des composants élémentaires. Dans l'exemple illustré par la Figure 5.2, l'instance `IC2` du composant `RC2` ne peut être activée avant que les données du port d'entrée `InPort3` soient présentes (contrainte 3). En d'autres termes, le port `InPort3` ne sera plein que quand le port `OutPort2` est à son tour plein. Nous déduisons ainsi les contraintes de dépendances de tâches suivantes :

- les neuf répétitions du composant `RC1` doivent être achevées avant l'exécution de la première répétition du composant `RC2` (contrainte 3);
- les neuf répétitions du composant `RC1` sont exécutées en parallèle (contrainte 1);
- les douze répétitions du composant `RC2` sont exécutées en parallèle (contrainte 1).

5.3.2 Modèle `pipeline`

Dans un modèle d'exécution `pipeline`, nous restreignons l'exécution parallèle à une exécution moins performante en nombre de processeurs virtuels. En effet, nous allouons pour chaque type d'instances de composants élémentaires un processeur virtuel. Ce processeur se voit confier l'exécution de toutes les répétitions d'une instance de composant. Donc le nombre de processeurs virtuels dans un tel choix d'exécution est égal au nombre d'instances de composants élémentaires, sans compter leurs espaces de répétitions. Le choix d'utiliser particulièrement ce type d'exécution permet de limiter le nombre de processeurs virtuels impliqués dans l'exécution. En effet il est parfois impossible

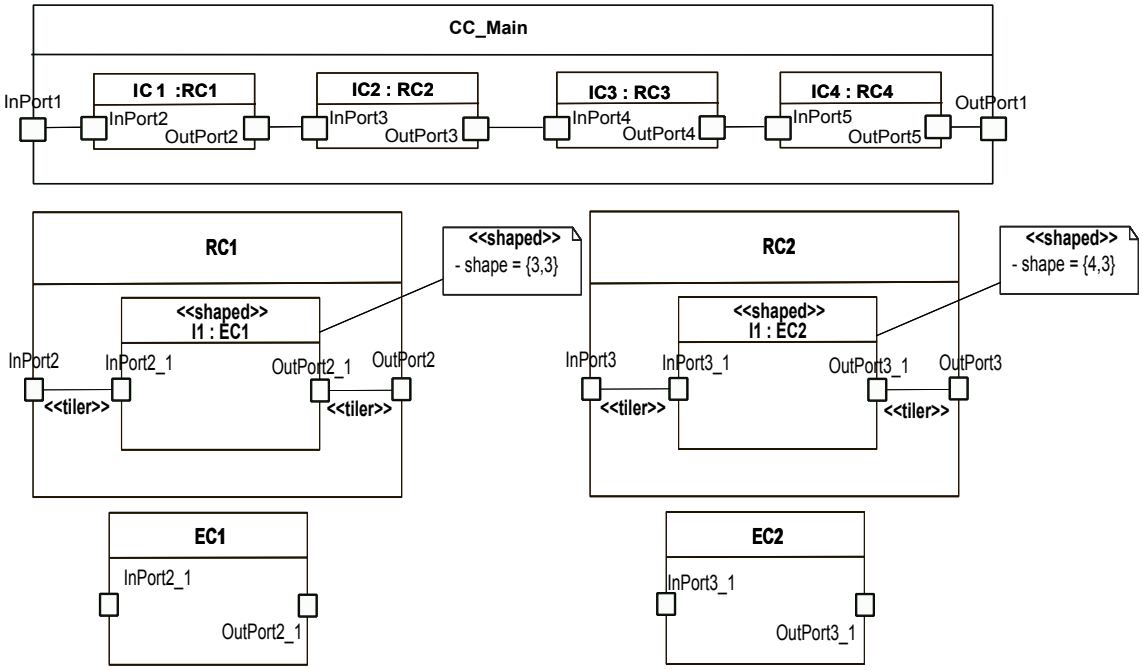


Figure 5.2: Exemple d'une hiérarchie de composants et d'instances de composants.

réellement d'allouer à chaque instance de composants un processeur virtuel, surtout quand l'espace de répétition des composants élémentaires est énorme (par exemple des centaines de millions de répétitions de composants élémentaires).

Les contraintes que nous déduisons à partir de ce modèle d'exécution sont les suivantes :

1. toutes les répétitions d'un composant élémentaire sont exécutées en séquentiel ;

2. deux composants élémentaires qui n'ont pas une dépendance directe entre eux sont exécutés en parallèle ;

3. quand deux composants élémentaires sont dépendants l'un de l'autre à travers des connecteurs, le composant producteur de données est exécuté avant ou au même moment que le composant consommateur.

Dans le cas de la Figure 5.2, deux processeurs virtuels sont considérés pour les composants élémentaires `EC1` et `EC2`. Le premier est alloué pour l'exécution séquentielle des neuf répétitions du composant `EC1` alors que le deuxième est alloué pour l'exécution séquentielle des douze répétitions du composant élémentaire `EC2` (contrainte 1). Selon la sémantique de RSM, une contrainte de précédence impose un ordre d'exécution du composant `RC1` avant `RC2`.

5.3.3 Modèle `mixte` par niveau d'hiérarchie

Dans un modèle d'exécution `mixte`, une exécution parallèle est considérée pour un groupe de composants élémentaires alors qu'une exécution pipeline est considérée pour un autre groupe. Nous essayons en cela de combiner les avantages des deux modèles précédents `parallèle` et `pipeline`.

Pour faire face à la complexité croissante de MPSoC évoquée à la fin du chapitre 2, surtout dans le cas d'applications hautes performances, nous avons proposé l'utilisation d'un modèle de calcul reposant sur le profil Marte. Ce modèle présente plusieurs avantages parmi lesquels :

- la description graphique de fonctionnalités sous forme de graphes acycliques de tâches ;
- la facilité du langage dans l'expression de parallélismes massifs de tâches et de données ;
- la conception à base de composants qui facilite la génération automatique de code vers différentes technologies cibles.

Nous avons ainsi répondu en partie à la question **Q1**, évoqué dans le chapitre 1.

5.4 Spécification des modèles d'exécution contraints par des rythmes d'interaction

Dans certains systèmes informatiques, il est souvent impossible d'avoir des horloges parfaitement synchronisées sur une échelle de temps globale. Parmi les solutions existantes, les horloges abstraites fournissent un moyen intéressant pour aborder ce problème. Une relation de précédence dite *"s'est produite avant"*, est définie pour capturer des dépendances causales entre des événements observés dans un système. Nous avons trouvé dans ce type d'horloges un moyen très performant pour décrire le comportement temporel d'une application modélisée en Marte.

À partir d'un modèle Marte, nous proposons une description de comportements fonctionnels à l'aide de traces d'horloges fonctionnelles. Ces horloges nous seront utiles, dans les chapitres 6 et 7, afin d'analyser des contraintes fonctionnelles (précédences et périodicités) et non fonctionnelles (temps d'exécution, consommation d'énergie). Ces travaux ont été publiés dans [5, 3, 6].

5.4.1 Horloges abstraites

Les horloges abstraites fonctionnelles que nous manipulons ont une notation binaire. Cette notation est inspirée des horloges k-périodiques définis par Cohen et al. [29]. Elles sont associées aux instances de composants élémentaires décrivant un comportement fonctionnel.

Definition 7 (Définition d'une horloge abstraite binaire) *Une horloge abstraite binaire est représentée par une séquence finie ou infinie de valeurs binaires,* 0 *et* 1*. Elle est associée à un composant de type élémentaire décrivant une fonctionnalité. L'occurrence d'une valeur "0" indique l'état inactif du composant alors que l'occurrence d'une valeur "1" indique son activation.*

À partir d'un modèle Gaspard2, nous souhaitons garder, dans une trace d'horloges fonctionnelles, les deux informations suivantes :

- *la quantité de travail* : cette information représente le *workload* que doit exécuter chaque processeur. À un niveau plus abstrait, cette information est donnée en

terme de nombre de composants élémentaires par processeur. À un niveau plus détaillé, elle est donnée en terme de nombre de cycles d'horloge processeur nécessaires pour l'exécution des fonctionnalités d'un composant élémentaire ;

- *les contraintes de précédences et de périodicités :* cette information dépend, d'une part, de la description fonctionnelle du système en Marte, et d'autre part, du choix du modèle d'exécution (`parallèle`, `pipeline` ou `mixte`). Nous souhaitons ainsi conserver l'ordre d'exécution des différentes tâches élémentaires.

Horloge abstraite périodique

Lors d'un traitement régulier de données, comme dans le cas des applications multimédias, le comportement fonctionnel est souvent périodique. Pour cela, nous définissons des horloges de type périodique. Ces horloges sont généralement compactes vues qu'elles sont souvent décrites par une période et une valeur initiale.

Definition 8 (Période) *Dans une horloge périodique, une période est le plus petit intervalle de temps qui se répète à plusieurs reprises.*

Soit *clk* une horloge abstraite périodique, elle peut être définie par :

$$clk = os(p)^r \tag{5.1}$$

où *os* (abréviation de *offset*) est la valeur initiale de l'horloge *clk* suivie de la période *p* qui est répétée *r* fois. Par exemple, si clk = 0 1 0 0 1 0 0 0 1 0 0 0, elle pourra être décrite d'une façon plus compacte telle que $clk = 0\ 1\ 0\ 0(1\ 0\ 0\ 0)^2$, où os = (0 1 0 0), p = (1 0 0 0) et r = 2.

5.4.2 Contraintes d'horloges abstraites

Le concept d'horloges abstraites se base sur l'occurrence d'événements représentés par les activations de composants. Selon la sémantique de UML, un composant modélisé au niveau application est activé quand il est instancié par le concept `instance specification`.

À chaque instant d'une horloge binaire clk_i portant la valeur "1", l'instance I_i du composant élémentaire C_i associée à celle-ci est activée. Durant chaque activation de C_i, les données, représentées sous forme de motifs, sont consommées par les ports d'entrée de I_i, traitées, puis produites et transmises à une autre instance I_j d'un composant C_j via les ports de sortie. À ce niveau de description, les latences de calculs de composants sont négligées.

Le nombre d'horloges fonctionnelles, dans une spécification Marte, peut varier d'une application à une autre. Cela dépend des contraintes imposées sur les composants élémentaires. Ces contraintes sont soit insérées manuellement par le concepteur, soit déduites du graphe de tâches fonctionnelles :

- des contraintes de périodicités et de précédences entre les activations des composants peuvent être insérées manuellement par le concepteur. Cela est possible en utilisant des concepts d'horloges synchrones définis dans les paquetages TIME et CCSL de Marte. Dans ce cas, le nombre d'horloges fonctionnelles dépend de la spécification des horloges et des contraintes d'horloges insérées par le concepteur ;

- quand le niveau de parallélisme d'un modèle est raffiné selon les trois modèles `parallèle`, `pipeline` ou `mixte`, le nombre d'horloges fonctionnelles ainsi que des contraintes de précédences et de périodicités sont déduites de la description de l'application (voir la section 5.3 pour plus d'informations sur ce sujet) :
  - pour un modèle d'exécution `pipeline`, le nombre d'horloges fonctionnelles est égal au nombre total d'instances de composants élémentaires sans compter leurs espaces répétitions ;
  - pour un modèle d'exécution `parallèle`, le nombre d'horloges fonctionnelles est égal au nombre de répétition totale de toutes les instances de composants élémentaires.

Considérons une instance `IC` d'un composant élémentaire `C` et *clk* = 010001001111 son horloge associée. Nous déduisons de cette horloge que le composant *C*, associé à l'horloge *clk*, est activé six fois. Le nombre d'activations est indiqué par le nombre d'occurrences de la valeur "1". Cependant l'occurrence de la valeur "0" dans une horloge binaire indépendante d'autres horloges n'a pas de sens. Pour cela, nous supposons l'existence d'une *horloge de référence*, généralement fictive, dans le système afin de synchroniser un ensemble d'horloges.

Definition 9 (Horloge de référence) *Une horloge de référence (ou horloge maîtresse), réelle (la plus rapide) ou fictive, permet de synchroniser les horloges esclaves sur son propre rythme. Les instants d'une horloge de référence sont les plus fréquents parmi les instants de toutes les autres horloges d'un système. Chaque occurrence d'un instant d'une horloge esclave est nécessairement superposable sur un instant de l'horloge de référence.*

L'horloge de référence nous servira de référence pour observer les occurrences de valeurs binaires sur les différentes horloges. En considérant plusieurs horloges fonctionnelles synchronisées sur une horloge de référence, l'occurrence d'une valeur 0 dans une horloge esclave implique des contraintes de dépendance de tâches (*s'est passé avant* ou *s'est passé après*). En effet, l'insertion de valeurs "0" dans une horloge permet de réorganiser la distribution des valeurs binaires par rapport aux instants de l'horloge de référence. Cela implique un nouvel ordre d'exécution de tâches.

En calant les différentes horloges fonctionnelles sur une horloge référence, nous conservons un ordre d'exécution des différentes tâches d'un modèle Marte. Ainsi, les contraintes de précédences et de périodicités, extrait d'un modèle Marte, sont préservées sur la trace d'horloges fonctionnelles correspondante.

Exemple de contraintes d'horloges abstraites fonctionnelles d'un modèle RSM

Les Figures 5.3, 5.4 et 5.5 montrent une modélisation d'une application modélisée Marte, respectivement avec les modèles d'exécutions `parallèle`, `pipeline` et `mixte`. Dans cette spécification, nous intégrons trois types de composants :

- le composant `Component B` est un composant du type composé. Il contient trois instances de composants `I1`, `I2` et `I3` ;
- le composant `Component A` est un composant du type répétitif. Il contient deux répétitions (cf. `shape={2,1}`) de l'instance `I0` du composant `Component B` ;
- les composants `Component C`, `Component D` et `Component E` sont trois composants élémentaires. Ils sont vides dedans et sont considérés comme étant des boîtes noires.

Selon cette configuration, les composants élémentaires `C`, `D` et `E` sont chacun répété deux fois dans `A`. Nous obtenons alors les couples d'instances (I_{1_1}, I_{1_2}), (I_{2_1}, I_{2_2}) et (I_{3_1}, I_{3_2}) associés respectivement aux composants `C`, `D` et `E` . Nous allons par la suite abstraire l'application, illustrée par les Figures 5.3, 5.4 et 5.5, selon trois traces d'horloges fonctionnelles. Les trois traces décrivent des contraintes de dépendances de tâches selon les modèles d'exécution `parallèle`, `pipeline` et `mixte`.

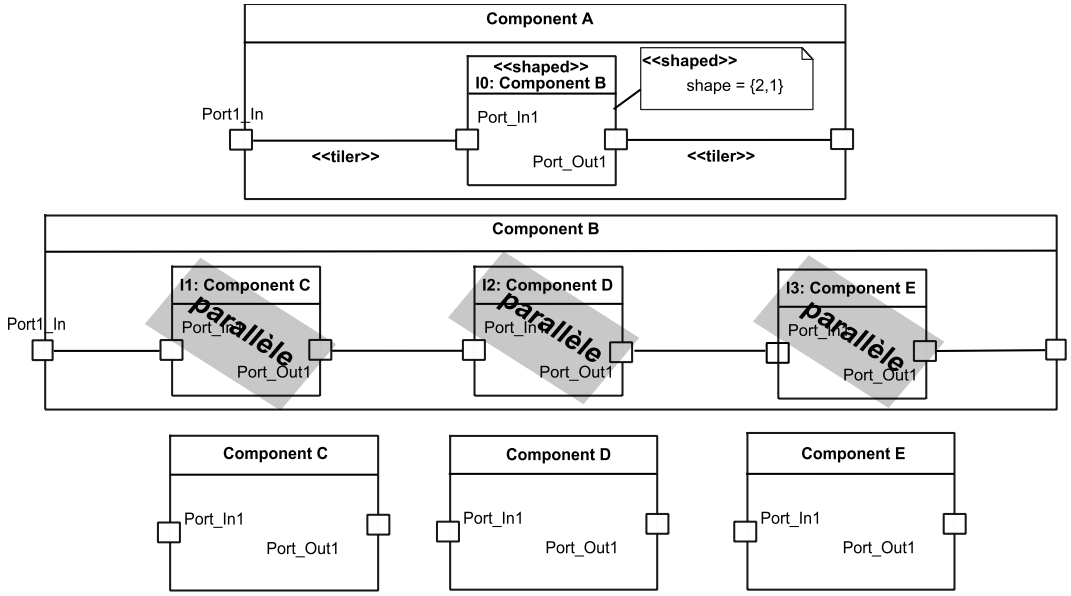


Figure 5.3: Exemple d'une application Marte avec une exécution parallèle.

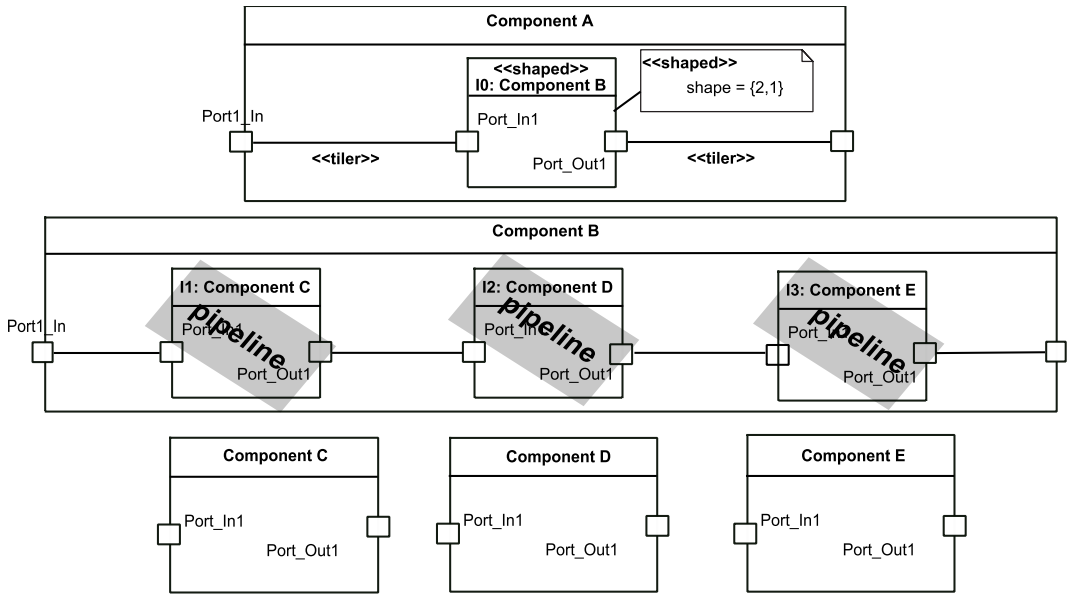


Figure 5.4: Exemple d'une application Marte avec une exécution pipeline.

La Figure 5.6 (a) montre une trace de six horloges binaires clk_{1_1}, clk_{1_2}, clk_{2_1}, clk_{2_2}, clk_{3_1} et clk_{3_2} associées respectivement aux instances de tâches I_{1_1}, I_{1_2}, I_{2_1}, I_{2_2}, $I3\_1$ et $I3\_2$. Cette trace abstrait l'application selon le modèle parallèle. Pour chaque répétition d'une instance de composant élémentaire, une horloge abstraite lui est associée. En

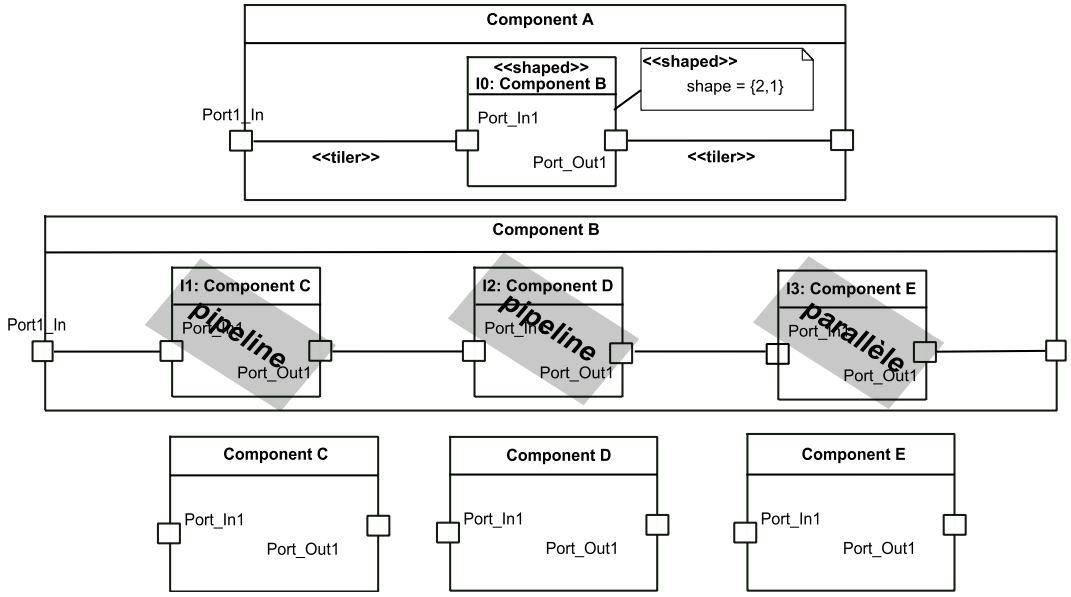


Figure 5.5: Exemple d'une application Marte avec une exécution mixte.

analysant les positions des occurrences des valeurs 1 sur chaque horloge (selon l'horloge de référence *IdealClk*) nous pouvons noter les constats suivants :

1. les instances I_{1_1} et I_{1_2} peuvent être exécutées en parallèle sans aucune contrainte de dépendance. En effet, les occurrences des valeurs 1, dans les horloges associées clk_{1_1} et clk_{1_2}, se superposent ;

2. le même constat que 1) est aussi valable pour les instances I_{2_1} et I_{2_2} d'une part et I_{3_1} et I_{3_2} d'autre part ;

3. une contrainte de précédence existe entre les horloges clk_{1_1} et clk_{1_2} d'une part, et les horloges clk_{2_1}, clk_{2_2} d'autre part. En effet, les instances I_{2_1} et I_{2_2} ne peuvent être activées avant les instances I_{1_1} et I_{1_2} ;

4. la même contrainte de précédence que 3) est valable pour les horloges clk_{2_1} et clk_{2_2}, d'une part, et les horloges clk_{3_1} et clk_{3_2} d'autre part.

IdealClk :	| | |
clk_{1_1}:	1 0 0
clk_{1_2}:	1 0 0
clk_{2_1}:	0 1 0
clk_{2_2}:	0 1 0
clk_{3_1}:	0 0 1
clk_{3_2}:	0 0 1

(a)

IdealClk :	| | | | | |
clk_1:	1 1 0 0 0 0
clk_2:	0 0 1 1 0 0
clk_3:	0 0 0 0 1 1

(b)

IdealClk :	| | | | |
clk_1:	1 1 0 0 0
clk_2:	0 0 1 1 0
clk_{3_1}:	0 0 0 0 1
clk_{3_2}:	0 0 0 0 1

(c)

Figure 5.6: Légende: (a) une trace d'horloges fonctionnelles selon le modèle d'exécution `parallèle`, (b) une trace d'horloges fonctionnelles selon le modèle d'exécution `pipeline` et (c) une trace d'horloges fonctionnelles selon le modèle d'exécution `pipeline` pour clk_1 et clk_2 et `parallèle` pour clk_3.

De la même manière, la Figure 5.6 (b) représente une trace d'horloges fonctionnelles selon le modèle d'exécution `pipeline`. Dans cette trace, seulement trois horloges

fonctionnelles clk_1, clk_2 et clk_3 sont définies. Ces horloges sont associées respectivement aux couples d'instances de tâches (I_{1_1}, I_{1_2}), (I_{2_1}, I_{2_2}) et (I_{3_1}, I_{3_2}). En analysant les positions des occurrences des valeurs 1 sur chaque horloge (selon l'horloge de référence *IdealClk*) nous pouvons noter les constats suivants :

- les deux instances I_{1_1} et I_{1_2} (resp. (I_{2_1}, I_{2_2}) et (I_{3_1}, I_{3_2})) du composant C_1 (resp. C_2 et C_3) sont exécutées en séquentiel ;
- l'activation d'une instance I_{1_1} (resp. I_{2_1}) précède l'activation de I_{2_1} (resp. I_{3_1}).

Dans la Figure 5.6 (c), nous avons choisis un modèle d'exécution `pipeline` pour les horloges clk_1 et clk_2 et le modèle d'exécution `parallèle` pour l'horloge clk_3. Nous pouvons aussi déduire un ordre d'exécution pour les différentes instances I_{1_1}, I_{1_2}, I_{2_1}, I_{2_2}, I_{3_1} et I_{3_2}.

5.4.2.1 Spécification de contraintes temporelles avec TIME et CCSL de Marte

Afin d'enrichir un modèle fonctionnel en Marte avec des informations temporelles, nous nous servons des paquetages CCSL et TIME de Marte, dont la sémantique est définie dans la section 4.3. Nous choisissons du paquetage TIME le concept d'horloges synchrones qui offre un support pour la modélisation des évènements de comparaison (avant, après ou simultanément) [4].

Dans un modèle Gaspard2, la modélisation de l'application est représentée dans un *diagramme de structure composite* proposé par UML 2.0 (également appelé *diagramme d'architecture*) [87]. Cette modélisation permet de décrire, sous forme de boîtes et flèches, des relations de dépendance de données entre différentes instances de composants.

La Figure 5.7, par exemple, définit respectivement 4 instances `IC1`, `IC2`, `IC3` et `IC4` des composants `AC1`, `AC2`, `AC3` et `AC4`. Toutes les instances sont contenues dans le composant `CC_Main` du type composé. Ces instances, une fois activées, consomment et génèrent des données à travers des ports d'entrées et de sorties.

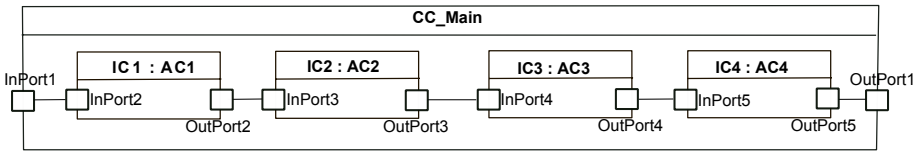


Figure 5.7: Spécification des fonctionnalités en Marte.

La Figure 5.8 montre des contraintes de périodicité écrite en CCSL et imposées sur des horloges de types synchrones. Sur le haut de la Figure 5.8, nous créons la classe `ComponentActivation` avec le stéréotype `ClockType`. Cette classe représente une horloge logique discrète dont l'unité est `ComponentActivationType` du type énumération. Nous créons ensuite quatre horloges logiques `clk1`, `clk2`, `clk3` et `clk4`. Ces horloges sont associées respectivement aux instances de classe `IC1`, `IC2`, `IC3` et `IC4`. Le type de ces horloges est `ComponentActivation`.

Après la création d'horloges associées aux instances de composants, nous décrivons maintenant le comportement temporel qui existe entre ces différentes horloges. Le bas de la Figure 5.8 montre trois contraintes d'horloges en CCSL. La première contrainte, par exemple, est liée à `clk1` et `clk4` (c'est l'attribue `on` qui précise les horloges liées à la contrainte). La contrainte décrit une relation de périodicité entre les horloges `clk1` et `clk4` dont la période et le décalage sont de quatre activations de l'instance `IC4`. Avec cette description dans un *diagramme de composant*, nous sommes en mesure de déduire la

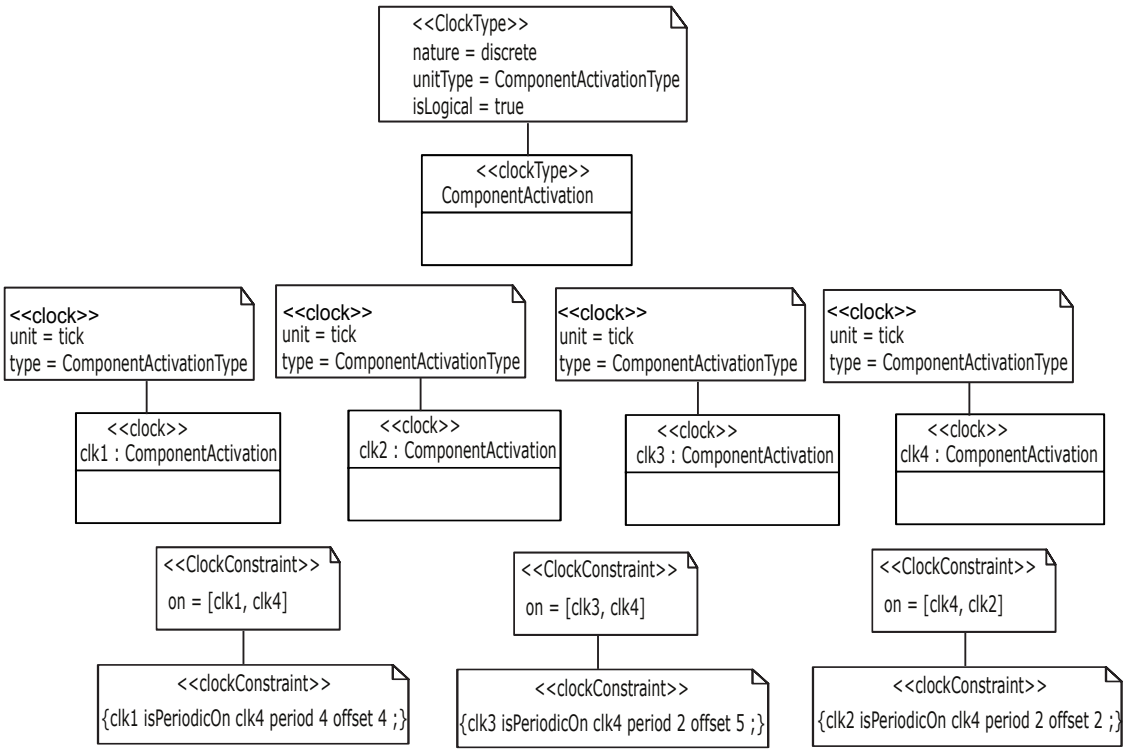


Figure 5.8: Contraintes temporelles spécifiées en Marte.

trace des instants de l'horloge `clk4` à partir de la trace de `clk1`. De même, la deuxième contrainte spécifie la relation de périodicité qui existe entre les horloges `clk3` et `clk4`.

5.5 Conclusion

Dans ce chapitre, nous avons montré comment certains concepts du profil Marte peuvent servir à modéliser des applications à un haut niveau d'abstraction. Nous avons utilisé le paquetage RSM afin de décrire des applications hautes performances nécessitant du parallélisme de tâches et de données répétitives. Nous avons aussi proposé une méthodologie pour abstraire le modèle de calcul initial selon les trois types de modèles : `parallèle`, `pipeline` et `mixte`. Cette abstraction de modèle est décrite par le biais d'horloges abstraites fonctionnelles dont la notation est inspirée des horloges k-périodiques. Elles nous serviront pour analyser formellement des propriétés fonctionnelles telles que la synchronisabilité de communication inter-tâches, les contraintes de précédences et de périodicités permettant de vérifier l'ordre d'exécution des tâches. Ces horloges nous servirons de même pour analyser des propriétés non fonctionnelles telles que l'estimation de performance et la consommation d'énergie du système.

Chapter 6

Utilisation de Marte pour la manipulation d'horloges abstraites

Nous proposons dans ce chapitre une méthodologie de conception d'architectures massivement parallèles et leurs associations sur des applications hautes performances. Cela nous ramène à une co-modélisation complète de systèmes comprenant de fonctionnalités allouées à des ressources physiques. Ensuite, nous abstrayons le système résultant par le biais d'horloges abstraites. Cette abstraction nous est utile pour analyser et vérifier des propriétés fonctionnelles, telles que l'ordre d'exécution de tâches fonctionnelles, et de propriétés non fonctionnelles, telles que le temps d'exécution et la consommation d'énergie.

Dans la section 6.1, nous modélisons, en Marte, une architecture massivement parallèle. Dans l'approche proposée, la modélisation d'architectures est complètement indépendante de la modélisation d'applications. Cela permet la réutilisation d'une architecture pour l'exécution de plusieurs modèles d'application. Dans la section 6.2, nous modélisons, en Marte, une association entre une architecture et une application. Ensuite, nous définissons, dans la section 6.3, une méthodologie d'abstractions de systèmes, co-modélisés en Marte, par le biais d'horloges abstraites.

6.1 Spécification d'architecture matérielle

Dans cette section, nous présentons une modélisation d'architectures matérielles à l'aide du profil Marte. Que se soit une architecture à mémoire partagée ou à mémoire

distribuée, des processeurs et des mémoires sont modélisés. Une architecture comprend toujours un ou plusieurs processeurs ayant des mémoires partagées ou distribuée.

6.1.1 Processeur et mémoire

Le domaine du matériel devient de plus en plus riche et varié. Il contient différents types d'architectures et de composants physiques. Par conséquent, la modélisation d'architectures nécessite un langage hautement expressif.

Dans Marte, le paquetage HRM décrit du matériel à travers plusieurs vues et niveaux de détails. Particulièrement, deux stéréotypes nous intéressent le plus :

- `HwProcessor` : il est défini dans le paquetage `HwComputing` et représente l'ensemble des ressources actives de calcul. Il représente, notamment, un processeur pouvant implémenter un ou plusieurs jeux d'instructions. Ce stéréotype contient plusieurs unités de gestion de mémoires et de caches organisées en différents types et niveaux ;

- `HwMemory` : ce stéréotype dénote une certaine quantité de mémoire. Elles peuvent être de types statiques (ROM–`HwROM`) ou dynamiques (RAM–`HwRAM`).

6.1.2 Exemple : modélisation d'une architecture PRAM

L'architecture que nous considérons doit être efficace et facile à configurer. Dans le but d'abstraire la complexité existante au niveau matériel, nous ne considérons que certains paramètres de configurations d'architecture afin d'atteindre les objectifs suivants :

- étudier l'impact du nombre de processeurs impliqués dans une exécution. Cela nous permet de trouver le nombre idéal de processeurs pour une architecture ;

- déterminer les valeurs de fréquences/voltages de processeurs. Cela nous permet de calculer la quantité d'énergie dynamique dissipée dans un système ;

- déterminer à quel moment un processeur doit être à l'état *actif* ou *éteint*. Quand un processeur est éteint, cela réduit la quantité d'énergie statique dissipée.

Dans le cadre de cette thèse, nous considérons des machines parallèles à accès aléatoire (*Parallel Random Access Machine*–PRAM). Ce type d'architecture est dédié au calcul parallèle. Chaque processeur dispose d'une horloge propre avec une certaine fréquence. À chaque *tick* de cette horloge, le processeur exécute un cycle d'instruction ou un fragment de cycle selon la taille et la complexité des instructions. La mesure du nombre de cycles par instruction sert à connaître le nombre moyen de cycles d'horloge requis pour qu'un processeur exécute une instruction.

Dans un cas particulier, nous considérons une machine CRCW (*Concurrent Read Concurrent Write*) qui permet des accès concurrents à une mémoire en écriture et en lecture. Les temps d'accès à la mémoire par des processeurs de types *Random Access Machine* (RAM) sont considérés uniformes.

La Figure 6.1 montre un exemple d'une architecture PRAM, modélisée en Marte. Cette architecture comporte trois processeurs `Processor1`, `Processor2` et `Processor3`. Ces derniers sont dédiés à l'exécution de composants fonctionnels. Les instances de composants `P1`, `P2` et `P3` sont stéréotypés tous `hwProcesseur` et ont respectivement les fréquences 45, 30 et 15 MHz. Les valeurs de fréquences sont spécifiées avec l'attribut `frequency` du stéréotype `hwProcessor`. Les processeurs communiquent directement

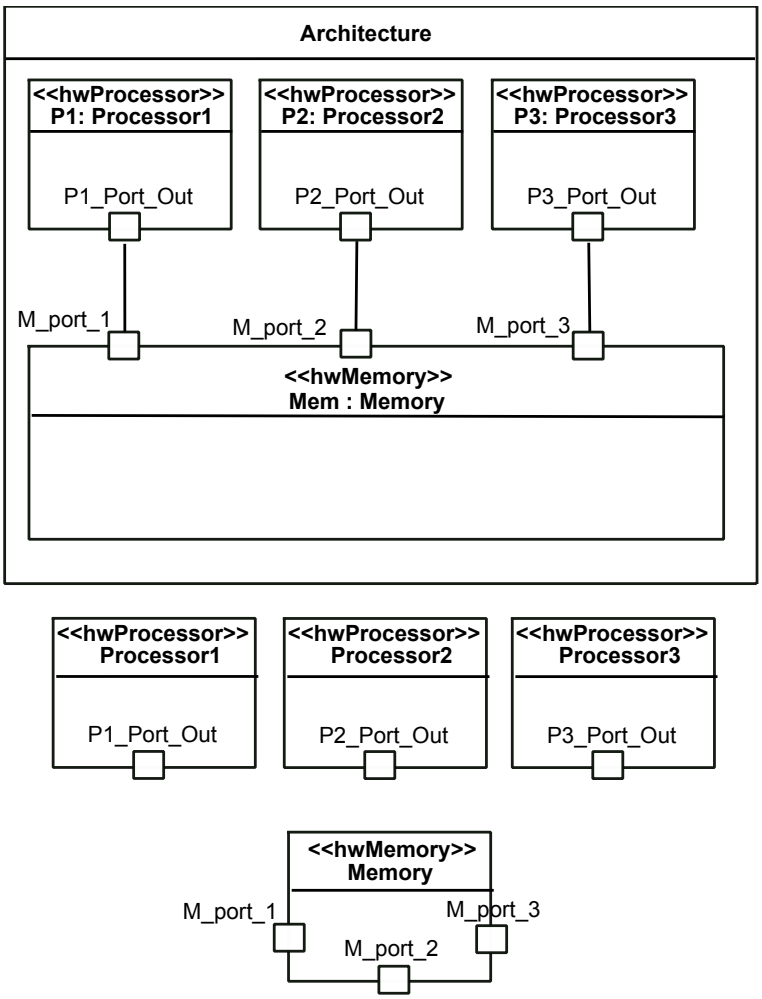


Figure 6.1: Modélisation Marte d'une architecture PRAM.

avec une mémoire partagée stéréotypée `hwMemory`. Cette mémoire permet d'écrire, stocker et récupérer des données et des instructions. Toutes les instances de composant, contenues dans l'architecture, sont stéréotypées avec des concepts du profil Marte (tous les noms commençant par `hw`).

> Nous avons vu dans cette section qu'il est possible de décrire graphiquement, et d'une façon compacte, des architectures massivement parallèles en se servant du profil Marte. Cela permet de répondre en partie à la question **Q1** posée dans le chapitre 1.

6.1.3 Représentation abstraite des fréquences de processeurs

Un processeur est formé d'un circuit électronique cadencé au rythme d'une horloge interne, que nous appelons horloge physique. Une telle horloge envoie des impulsions, appelées *top*. La fréquence d'un processeur, exprimée en Hertz (Hz), correspond au nombre de *top* par seconde. Ainsi, la période d'un processeur est égale à l'inverse de sa fréquence. Par exemple, un processeur ayant une fréquence de 100 Hertz possède une période égale à 0.01 seconde.

À chaque top d'horloge, un processeur exécute une instruction ou une partie d'instruction. L'indicateur appelé *Cycles Par Instruction* (CPI) représente le nombre moyen de cycles d'horloge nécessaire, pour l'exécution d'une instruction sur un processeur. La puissance d'un processeur peut ainsi être caractérisée par le nombre d'instructions qu'il est capable de traiter par seconde; l'unité utilisée étant le MIPS (*Millions d'Instructions Par Seconde*).

6.1.3.1 Horloge de référence

En considérant les périodes d'un ensemble de processeurs impliqués dans une exécution d'une fonctionnalité, nous pouvons tracer les cycles d'horloge pour chaque processeur. Cependant, l'utilisation de plusieurs horloges physiques dans un système peut provoquer des problèmes de synchronisation d'horloges. Cela est dû aux dérives d'horloges qui peuvent causer des décalages entre différents instants superposables de différentes horloges. Par conséquent, des points de synchronisation doivent être ajoutés afin de préserver l'ordre partiel d'évènements comme dans Lamport et al. [69]. Quand plusieurs processeurs sont considérés dans une architecture, une horloge de référence est nécessaire afin de synchroniser les communications inter-processeurs.

Afin de synchroniser les différentes horloges d'un système, nous considérons donc une horloge de référence ayant une base de temps commune pour toutes les autres horloges. Nous définissons ainsi une horloge de référence fictive, appelée *IdealClk*, qui est la plus fréquente parmi toutes les autres horloges.

6.1.3.2 Analyse de fréquences

Étant donné une architecture multiprocesseur, nous considérons un ensemble $P = \{Proc_i, \forall i \in [1, n]\}$ de n processeurs. Pour chaque processeur $Proc_i$, une fréquence f_i lui est associée à travers le stéréotype `hwprocessor` de Marte. Nous définissons ainsi $d_i = 1/f_i$ comme étant la période entre deux cycles d'horloge successifs durant l'activité du processeur $Proc_i$.

Chacun des processeurs $Proc_i$ de P possède une horloge physique $PhysicalClk_i$ décrivant sa vitesse de calcul. Elle est représentée par un ensemble d'instants successifs. Afin de caractériser l'horloge de référence $IdealClk$ de l'ensemble P des processeurs, les horloges physiques des différents processeurs doivent vérifier la propriété suivante :

$$IdealClk \supseteq PhysicalClk_i, \forall i \in [1, n] \tag{6.1}$$

La fréquence correspondante à $IdealClk$ peut être calculée de plusieurs manières. Nous proposons de la définir comme suit :

$$f_{IdealClk} = PPCM(f_i), \forall i \in [1, n] \tag{6.2}$$

où PPCM représente le calcul du plus petit commun multiple. Cela revient à calculer le plus petit commun multiple de l'ensemble des fréquences associées aux différents processeurs impliqués dans l'exécution comme dans Nielsen et al. [83]. Le calcul du PPCM n'est pas la solution optimale. Il existe de nombreuses autres techniques. Cependant, le PPCM assure toujours l'obtention d'une valeur de fréquence pour une horloge de référence.

6.1.3.3 Exemple

Prenons l'exemple d'une architecture multiprocesseur illustrée par la Figure 6.1. Cette architecture incorpore trois processeurs P_1, P_2 et P_3 ayant respectivement les fréquences 30, 45 et 15 MHz. Afin de tracer les instants logiques de l'horloge de référence $IdealClk$, nous calculons le plus petit commun multiple de ces trois valeurs de fréquence : $PPCM(30, 45, 15) = 90$. Cela veut dire que pour chaque 30 cycles (resp. 45 et 15 cycles) d'horloge du processeur P_1 (resp. P_2 et P_3), l'horloge de référence fera 90 tops d'horloge. Ainsi, la période de l'horloge référence est : $P_{IdealClk} = \frac{1}{90} \sim 0.011$ microsecondes.

Figure 6.2: Horloges physiques associées aux processeurs P_1, P_2 et P_3.

La Figure 6.2 illustre la relation de dépendance qui existe entre l'horloge de référence $IdealClk$ et les horloges $PhysicalClk_1$, $PhysicalClk_2$ et $PhysicalClk_3$ associées respectivement aux processeurs P_1, P_2 et P_3. Pour chaque cycle d'horloge du processeur P_3, il y a trois cycles d'horloge du processeur P_2 et deux cycles d'horloge du processeur P_1.

6.2 Association en Marte

Une fois que les fonctionnalités d'un algorithme et l'architecture matérielle correspondante sont modélisées, les éléments décrivant des fonctionnalités sont alloués sur des ressources physiques de l'architecture. Cette phase est appelée association ou allocation.

Definition 10 (Association) *L'association d'une application à une architecture consiste à allouer des composants fonctionnels dans la partie application aux ressources physiques correspondantes dans la partie architecture.*

Durant cette phase, il est nécessaire de connaître le nombre de cycles d'horloge processeur nécessaire à l'exécution d'une instance de composant fonctionnel. Cette information peut varier selon le type de processeur et le jeu d'instruction choisi. Le nombre de cycle d'horloge pour l'exécution d'une instance de composant élémentaire nous sera utile dans le chapitre 7 afin de tracer l'activité des processeurs en terme de cycles d'horloge.

En Marte, c'est avec les stéréotypes `allocate` et `distribute` que l'utilisateur exprime l'association entre une application et une architecture. Dans la Figure 6.3 par exemple, les instances `I1`, `I2` et `I3` des composants `C`, `D` et `E`, contenues dans la description fonctionnelle du système, sont associées respectivement aux processeurs `P1`, `P2` et `P3`. Cette association est exprimée par le biais de trois connecteurs, de type `abstraction` (un concept dans UML), tous stéréotypés `allocate`.

Jusqu'à présent, nous nous placions à un niveau de modélisation indépendant d'une plateforme d'exécution. Par la suite, il s'agit de raffiner les modèles décrits à ce niveau vers des implémentations spécifiques selon le besoin d'un concepteur : simulation à différents niveaux d'abstraction en SystemC, synthèse d'accélérateurs matériels à l'aide de VHDL, etc. Afin de cibler chacune des implémentations possibles dans Gaspard2, le concept de propriété intellectuelle (Intellectual Property, IP) est utilisé : c'est la phase de déploiement[1].

Une des questions fondamentales posées dans le chapitre 1 est la question **Q2**. Cette question s'interroge sur la manière d'accélérer la conception d'applications hautes performances sur des architectures massivement parallèles. Notre travail répond à cette question en proposant un environnement de conceptions de MPSoC, à un haut niveau d'abstraction, en se basant sur le profil Marte. Cet environnement rend facile la conception de MPSoC dans un temps réduit. Cela est dû, d'une part, à l'utilisation d'un langage de modélisation graphique agréable et facile à programmer, et d'autre part, à la puissance du modèle de calcul sous-jacent pour exprimer le parallélisme de tâches et de données. Mener à cet environnement, un utilisateur n'a qu'à suivre trois étapes afin de modéliser différentes configurations de systèmes. Les étapes consistent à modéliser des fonctionnalités, fixer les paramètres d'une architecture multiprocesseur et modéliser une ou plusieurs associations possibles de tâches fonctionnelles sur des ressources physiques. Dans une étape suivante, nous proposons une méthodologie d'analyse et de vérification de systèmes à base d'horloges abstraites. Cette méthodologie, faite à un haut niveau d'abstraction, permet de réduire l'espace de solutions des différentes configurations possibles d'architectures et d'associations.

6.3 Modélisation de l'exécution à l'aide d'horloges abstraites

Dans la section 5.4, nous avons défini des horloges abstraites décrivant le comportement fonctionnel d'un système. De la même manière, dans la section 6.1.3 nous avons déduit des horloges abstraites physiques qui décrivent les vitesses d'exécutions des processeurs d'une architecture multiprocesseur. Dans cette section, nous proposons une projection d'horloges abstraites fonctionnelles sur des horloges abstraites physiques afin de caractériser l'exécution d'un système sous forme d'une trace d'horloges abstraites simulant

[1] Le lecteur peut se référer au chapitre 4 pour plus de détails concernant la modélisation du déploiement.

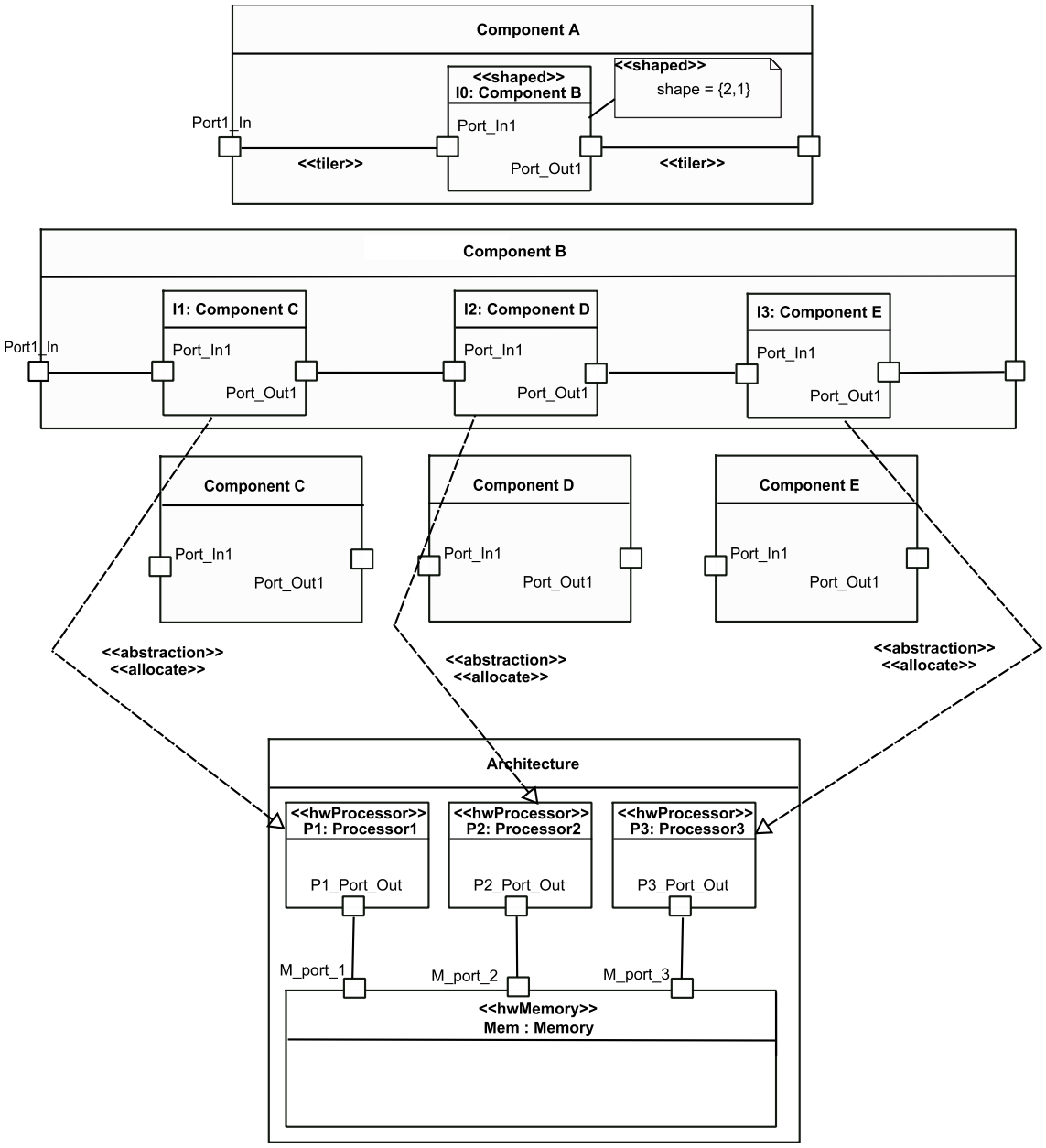


Figure 6.3: Modélisation Marte de l'association entre des instances de composants fonctionnels et des processeurs.

l'activité des processeurs durant l'exécution de fonctionnalités. Nous appelons "*horloges d'exécutions*" ce type d'horloges.

Nous proposons dans cette section un algorithme d'ordonnancement, du type statique non préemptif, de tâches fonctionnelles sur des processeurs.

Un ordonnancement correct ne dépend pas uniquement du résultat fonctionnel de l'exécution de tâches, c'est-à-dire des valeurs calculées, mais aussi du respect des échéances temporelles prévues. Le but est donc de définir un ordre d'exécution des différentes tâches fonctionnelles qui préservent les contraintes de dépendances de tâches et les temps d'échéances imposés sur l'application. Cet algorithme, appliqué sur des horloges abstraites fonctionnelles et physiques, engendre une trace d'horloges abstraites d'exécutions simulant l'activité des différents processeurs.

6.3.1 Projection d'une horloge fonctionnelle sur une horloge physique

Nous considérons dans cette section le cas d'un ordonnancement mono-tâche/mono-processeur. Cet ordonnancement est réalisé par le biais d'horloges abstraites. En entrée, nous considérons :

- une horloge fonctionnelle clk_i, représentant les activations d'une tâche T_i par rapport à une horloge de référence ;
- une horloge physique $PhysicalClk_i$ associée à un processeur $Proc_i$ représentant les cycles d'horloge de ce dernier ;
- une latence de calcul c_i représentée par le nombre de cycles horloge de $Proc_i$ nécessaires pour l'exécution d'une instance de la tâche T_i.

Après la phase d'association, nous redistribuons les instants de l'horloge clk_i sur les instants de l'horloge $PhysicalClk_i$, en prenant en compte les latences réelles de calcul c_i.

Les horloges abstraites clk_i et $PhysicalClk_i$ sont respectivement associées à deux variables λ et $\phi(\lambda)$, telles que:

- $\lambda \in [1, taille(clk_i)]$ indique une position d'un instant de clk_i ;
- $\phi(\lambda) \in [1, taille(PhysicalClk_i)]$ indique la position d'un instant logique λ de clk_i sur l'horloge $PhysicalClk_i$.

La position $\phi(\lambda)$ est calculée en fonction de λ et de la valeur portée par les instants dans clk_i selon l'algorithme ***Mono_Scheduling*** définis dans la Figure 6.4.

Pour chaque instant λ dans clk_i, la fonction $\phi(\lambda)$ est calculée récursivement selon la formule suivante :

$$\phi(\lambda) = \begin{cases} \phi(\lambda - 1) + c_i & si\ clk_i[\lambda - 1] == 1 \\ \phi(\lambda - 1) + 1 & si\ clk_i[\lambda - 1] == 0 \end{cases} \tag{6.3}$$

Pour tout $\lambda \in [1, n]$, les valeurs binaires (0 ou 1), ayant les position λ sur l'horloge clk_i, sont projetées sur les instants $\phi(\lambda)$ de l'horloge $PhysicalClk_i$. Une fois toute les valeurs binaires projetées, chaque instant de l'horloge $PhysicalClk_i$, ne portant pas une valeur binaire, lui sera affecté une valeur -1.

Nous obtenons ainsi une nouvelle horloge abstraite $Tclk_i$ dont les valeurs sont de types ternaires. Cette horloge est le résultat de la projection des instants binaires de clk_i sur les instants de l'horloge physique $PhysicalClk_i$ (cf. Figure 6.5).

```
Mono_Scheduling(clk_i, c_i, PhysicalClk_i) () {
Int Taille := taille(clk_i);        /*On calcule la taille de clk_i.*/
φ(1) := 1;        /*On commence l'ordonnancement par le premier instant de clk_i.*/
for λ ∈ [2, Taille] do
  if clk_i[λ − 1] == 1        /*Si l'instant précédent de clk_i porte la valeur 1.*/ then
    φ(λ) := φ(λ − 1) + c_i;    /*On avance de c_i instants.*/
  else if clk_i[λ − 1] == 0        /*Si l'instant précédent de clk_i porte la valeur 0.*/
  then
    φ(λ) := φ(λ − 1) + 1;        /*On avance d'1 instant.*/
  end if
end for
```

Figure 6.4: Algorithme d'ordonnancement mono-tâche/mono-processeur.

Quand un instant de l'horloge clk_i porte la valeur "1", il est projeté sur l'horloge $PhysicalClk_i$ suivi de $(c_i - 1)$ instants portant la valeur "-1". Ces instants marquent les latences réelles de calcul de la tâche T_i. En introduisant le symbole "-1" dans l'horloge $Tclk_i$, nous gardons la même sémantique des occurrences des valeurs "0" et "1" que dans les horloges abstraites fonctionnelles. Chaque instant de l'horloge ternaire $Tclk_i$ portera une des trois valeurs suivantes :

- *"1"* : le processeur $Proc_i$ associé à la tâche T_i est en mode *actif*. À cet instant logique, il est entrain d'exécuter une instruction entière ou un fragment d'une instruction ;
- *"0"* : la valeur 0 est insérée pour préserver l'ordre d'exécution des tâches. Le processeur $Proc_i$ associé à la tâche T_i est en mode *No Operation* (*NOp*). Cette valeur indique l'inactivité due aux problèmes de dépendances entre des tâches communicantes ;
- *"-1"* : l'occurrence d'une valeur "-1" est cependant contextuel. Étant donné une suite s de valeur "-1" :
  - si s est immédiatement précédée par 1, alors s indique le mode *potentiellement actif* du processeur associé ;
  - sinon, s indique le mode *idle* du processeur associé due à un processus de synchronisation.

Les horloges d'exécutions, obtenues à partir d'une projection d'horloges fonctionnelles sur des horloges physiques, offrent un support d'analyse et de vérification de l'ordre d'exécution de tâches. Cet ordre d'exécution est imposé durant la modélisation de fonctionnalités, sous forme de contraintes de précédences et de périodicités de tâches.

Afin de faciliter la compréhension de cet algorithme, nous montrons dans la Figure 6.5 une trace détaillée de la projection d'une horloge fonctionnelle clk_i sur une horloge physique $PhysicalClk_i$. L'horloge d'exécution $Tclk_i$, résultante de cet ordonnancement, représente l'activité du processeur $Proc_i$ durant l'exécution de la tâche T_i.

6.3.1.1 Exemple de projection d'horloges

Nous appliquons l'algorithme d'ordonnancement mono-tâche/mono-processeur sur trois horloges abstraites clk_1, clk_2 et clk_3. Ces horloges, illustrées par la Figure 6.6 (a), ont respectivement les latences de calculs $c_1 = 4$, $c_2 = 2$ et $c_3 = 1$. Nous pouvons déduire,

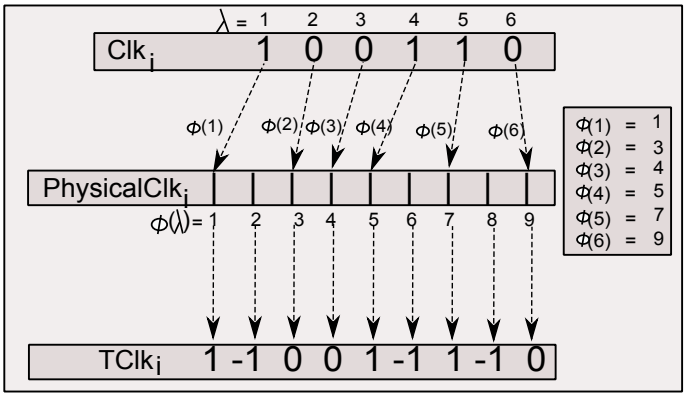


Figure 6.5: Projection des instants de l'horloge fonctionnelle clk_i sur l'horloge physique $PhysicalClk_i$, avec $c_i = 2$.

à partir de cette trace d'horloges, deux informations essentielles de la description des fonctionnalités :

- *la quantité de travail* : en analysant le nombre d'occurrences des valeurs "1", nous pouvons savoir le nombre d'exécution du composant associé ;
- *les contraintes de périodicités et de précédences* : en analysant les positions des instants, portants la valeur "1" dans les différentes horloges, nous pouvons déduire des contraintes de précédences et de périodicités sur l'exécution des différents composants.

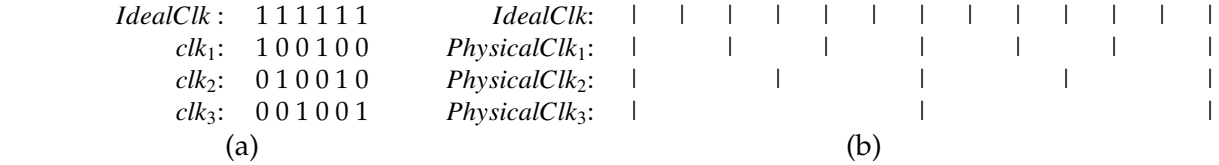


Figure 6.6: Légende: (a) une trace de trois horloges abstraites fonctionnelles et (b), une trace de trois horloges abstraites physiques selon une horloge de référence.

Afin de projeter les instants des trois horloges clk_1, clk_2 et clk_3 sur des horloges physiques, nous considérons la trace d'horloges physiques obtenue dans la section 6.1.3. Les horloges physiques $PhysicalClk_1$, $PhysicalClk_2$ et $PhysicalClk_3$ (cf. Figure 6.6 (b)) sont associées respectivement aux tâches clk_1, clk_2 et clk_3. Ces horloges représentent les vitesses de traitement des trois processeurs P_1, P_2 et P_3.

La Figure 6.7 montre une trace de trois horloges d'exécutions $Tclk_1$, $Tclk_2$ et $Tclk_3$. Ces dernières sont le résultat de l'ordonnancement des horloges fonctionnelles clk_1, clk_2 et clk_3 respectivement sur les horloges physiques $PhysicalClk_1$, $PhysicalClk_2$ et $PhysicalClk_3$. Nous remarquons que l'information concernant la quantité de travail (représentée par l'occurrence des 1 dans les horloges binaires) est conservée.

Cependant, les contraintes de dépendances de tâches, contenues dans la trace d'horloges fonctionnelles, doivent être vérifiées. En effet, en redistribuant les valeurs "0" et "1" des horloges fonctionnelles, la nouvelle trace obtenue des horloges d'exécutions pourra contenir des violations de l'ordre d'exécution de tâches, imposé par les contraintes de précédences et de périodicités. Pour cela, une étape de vérification des ces contraintes est définie dans le chapitre 7.

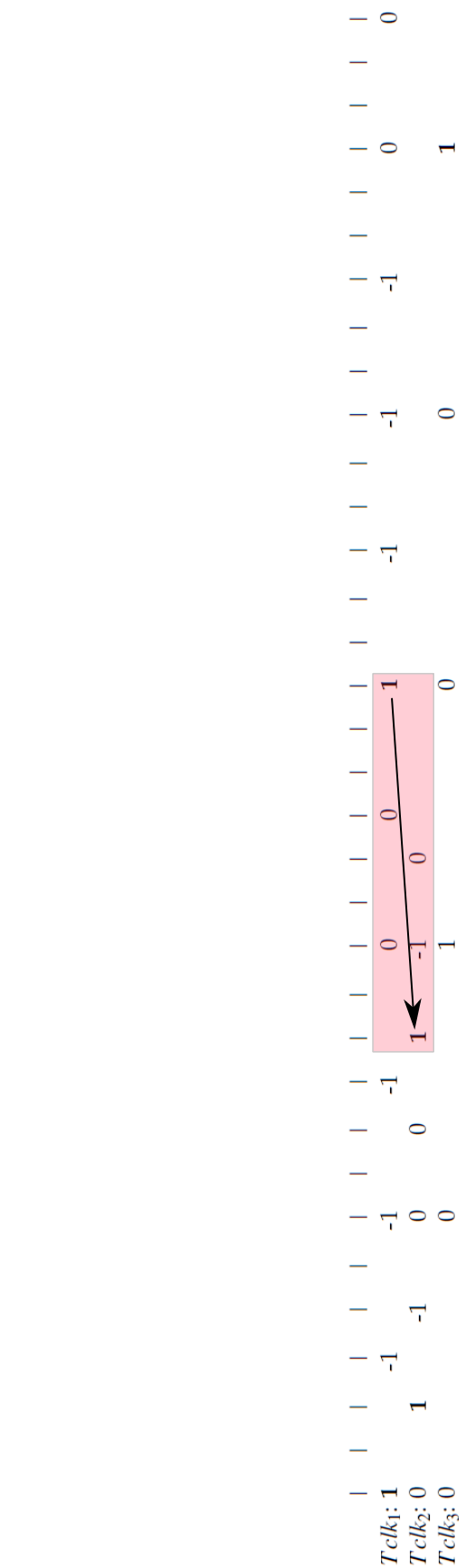


Figure 6.7: Trace de simulation des trois horloges abstraites $Tclk_1$, $Tclk_2$ et $Tclk_3$.

Dans la section suivante, nous présentons un algorithme d'ordonnancement de plusieurs horloges abstraites fonctionnelles sur une horloge abstraite physique. Cela représente le cas d'un ordonnancement de plusieurs tâches fonctionnelles sur un seul processeur.

6.3.2 Projection de plusieurs horloges fonctionnelles sur une horloge physique

Dans cette section, nous traitons le cas d'une association d'un ensemble de tâches fonctionnelles $T = \{T_1, T_2, ..., T_n\}$ sur un seul processeur que nous appelons *Proc*. Nous considérons un ensemble C d'horloges abstraites fonctionnelles totalement ordonnées :

$$C = \{clk_1, clk_2, ..., clk_n\} \tag{6.4}$$

où n représente le nombre total d'horloges fonctionnelles. Chaque horloge clk_i de C représente l'activation de plusieurs instances d'une tâche T_i.

Dans la solution proposée, chaque horloge clk_i de C est divisée en m tranches T_i^j de même taille, où $i \in [1, n]$ et $j \in [1, m]$. *Proc* commute ainsi son exécution sur les différentes tranches T_i^j des différentes horloges clk_i.

L'exécution d'une tâche fonctionnelle nécessite un certain nombre de cycle processeur. Pour cela, nous définissons l'ensemble CP, totalement ordonné, et contenant le nombre de cycles processeur c_i nécessaires pour l'exécution des différentes tâches T_i associées aux horloges clk_i, où $i \in [1, n]$:

$$CP = \{c_1, c_2, ..., c_n\} \tag{6.5}$$

Nous considérons aussi une horloge physique *PhysicalClk* associée à *Proc*.

Nous définissons dans la Figure 6.9 un algorithme d'ordonnancement, de type statique non préemptif, de tâches fonctionnelles sur un processeur. Cet algorithme manipule des horloges fonctionnelles clk_i (représentant des tâches fonctionnelles) et une horloge physique *PhysicalClk* (représentant la vitesse d'un processeur). En entrée, nous considérons :

- un ensemble de n horloges fonctionnelles clk_i, représentant les activations de tâches T_i par rapport à une horloge de référence, où $i \in [1, n]$;
- une horloge physique $PhysicalClk_i$ associée à un processeur $Proc_i$ représentant les cycles d'horloge de ce dernier ;
- un ensemble de valeur c_i représentant les latences de calcul des différentes tâches fonctionnelles, représentées par le nombre de cycles horloge de $Proc_i$ nécessaires pour l'exécution d'une instance de la tâche T_i, où $i \in [1, n]$.

Cet algorithme permet de calculer la position ϕ sur *PhysicalClk*, d'une projection d'un instant λ d'une horloge fonctionnelle clk_i. De la même manière que l'ordonnancement mono-tâche/mono-processeur, quand un instant d'une horloge clk_i porte la valeur "1", il est projeté sur l'horloge $PhysicalClk_i$ suivi de $(c_i - 1)$ instants portant la valeur "-1". Le résultat de cet ordonnancement est une horloge d'exécution ayant des valeurs ternaires. Cette horloge simule l'activité du processeur durant l'exécution des fonctionnalités.

L'activité du processeur *Proc* est contrôlée par un pointeur, représentant un ordonnanceur. Dans cet algorithme, *Proc* commence l'exécution par la première tranche T_1^1 de la tâche T_1 associée à l'horloge clk_1. L'ordonnanceur vérifie d'abord la valeur du premier instant dans T_1^1 :

- si la valeur est "1", alors le premier instant de *PhysicalClk* portera la valeur "1" et le pointeur est initialisé à la position (c_1) de l'horloge *PhysicalClk* ;
- si la valeur est "0", alors le premier instant de *PhysicalClk* portera la valeur "0" et le pointeur est initialisé à la position 2 de l'horloge *PhysicalClk*.

Ensuite, l'ordonnanceur exécute récursivement cette étape sur le reste des valeurs binaires de la tranche T_1^1. Quand la valeur rencontrée est "1", le pointeur avance de c_1 instant sur *PhysicalClk*. Sinon, le pointeur avance d'un instant logique.

Une fois la tranche T_1^1 projetée sur *PhysicalClk*, l'ordonnanceur alloue le processeur pour l'exécution de la première tranche T_2^1 de l'horloge fonctionnelle clk_2 et ainsi de suite jusqu'à la projection de toutes les tranches T_i^j des horloges clk_i sur les instants de *PhysicalClk*, où $i \in [1, n]$ et $j \in [1, m]$.

La Figure 6.8 montre la projection des trois horloges clk_1, clk_2 et clk_3 (cf. Figure 6.6 (a)) sur une horloge physique $PhysicalClk_i$. L'horloge clk_1 (resp. clk_2 et clk_3) est découpée en tranches de taille deux (resp. trois et six) et possède un cycle d'horloge $c_1 = 4$ (resp. $c_2 = 2$, $c_3 = 1$) pour chaque activation d'une tâche. L'ordonnanceur exécute les différentes tranches des différentes horloges selon l'ordre suivant : T_1^1, T_2^1, T_3^1, T_1^2, T_2^2, T_1^3.

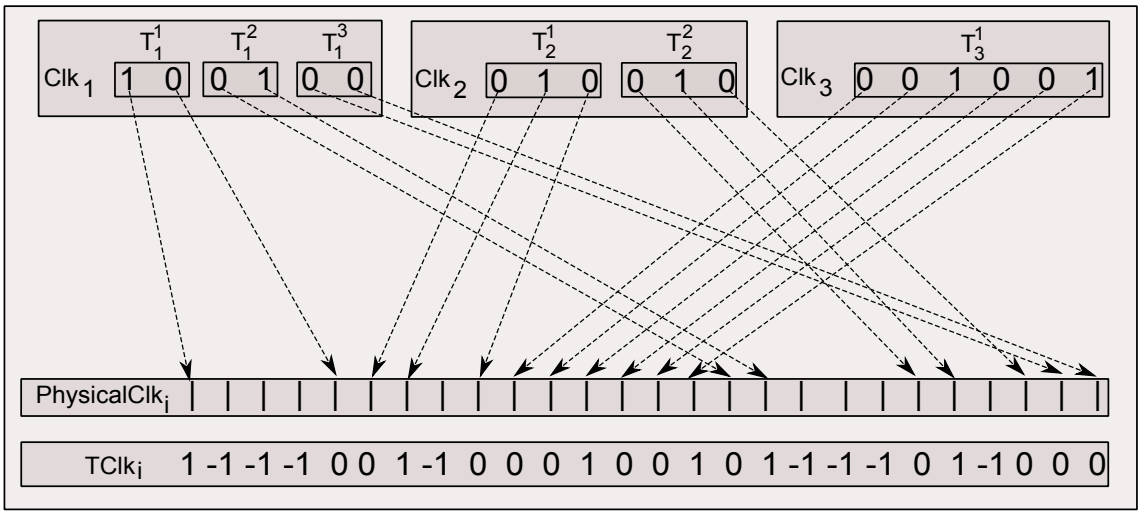


Figure 6.8: Projection des instants de trois horloges fonctionnelles clk_1, clk_2 et clk_3 sur une horloge physique $PhysicalClk_i$, avec $c_1 = 4$, $c_2 = 2$ et $c_3 = 1$.

6.3.3 D'autres scénarios de projections d'horloges abstraites

Dans cette section, nous discutons les deux cas d'ordonnancements suivants : mono-tâche/multiprocesseur et multi-tâches/multiprocesseur. Ces deux cas d'ordonnancement peuvent être effectivement réduit en terme de scénarios d'ordonnancement mono-tâche/mono-processeur ou multi-tâches/mono-processeur présentés respectivement dans la section 6.3.1 et 6.3.2.

Typiquement, ce type d'ordonnancement est utile quand le concepteur souhaite une exécution parallèle des différentes tâches fonctionnelles sur différents processeurs. Afin d'allouer une tâche T à n processeurs, nous partitionnons la tâche T en n sous-tâches formant ainsi l'ensemble $E_T = \{T_1, ..., T_n\}$. Ainsi, nous associons aux différentes sous-tâches de E_T des processeurs. En faisant ainsi, nous revenons au cas d'un ordonnancement

Var V'_c; /*Le nombre de cycle d'instruction correspondant à la dernière valeur binaire placé sur PhysicalClk.*/
Var $\{S_1, S_2, ..., S_n\}$:=$\{0, 0, ..., 0\}$; /*Ensemble des pointeurs, initialisées à 0, indiquant l'instant de la fin du travail d'un processeur sur les horloges logique. */

Var $Tab[n] := [1, 2, ..., n]$; /*Tableau trié contenant les indices des horloges en cours d'exécutions.*/
Var $id[n] := [1, 2, ..., n]$; /* Tableau trié contenant les identifiants des différentes horloges logiques.*/

- -

Ordonnancement_Multi_tâches() {
V_c :=*InitialiserV*$_c$(); /*Appelle à la fonction *InitialiserV*$_c$.*/
$id := 1$; /*On commence l'ordonnancement par la première horloge.*/
while $Tab[i] \neq 0$ /*Tant qu'il y a des tranches à placer sur PhysicalClk.*/ **do**
  $(S_{id}, V'_c) := Mono\_Slicing(S_{id}, c_{id}, V'_c)$; /*Placer la valeur binaire $clk_{id}[S_{id}]$ sur PhysicalClk.*/
end while
}

- -

InitialiserV_c() { /*Fonction qui initialise la valeur V_c .*/
if $clk_1[1]$==1 /*Si le premier instant de clk_1 porte la valeur 1.*/ **then**
  $V_c := c_1$; /*Le pointeur avance de c_1 instants.*/
else if $clk_1[1]$==0 /*Si le premier instant de clk_1 porte la valeur 0.*/ **then**
  $V_c := 1$; /*Le pointeur avance d'1 instant.*/
end if
Return V_c; /*Retourne la valeur V_c.*/
}

- -

Mono_Slicing(S_{id}, c_{id}, V'_c) () {
Char $Sequence[]$:=$[clk_{id}[S_{id}+1], clk_{id}[S_{id}+2], ..., clk_{id}[S_{id}+s_i]]$;
if $Sequence \equiv \emptyset$ /*S'il n'existe plus de tranche à placer.*/ **then**
  $Tab[id] := 0$; /*Supprimer l'identifiant de l'horloge de Tab.*/
else if $Sequence \neq \emptyset$ **then**
  for $\lambda \in [S_{id}, S_{id} + s_{id}]$ /*Parcours de la tranche.*/ **do**
    $\phi(\lambda) := \phi(\lambda - 1) + V'_c$; /*Le pointeur avance de V'_c instants.*/
    if $Sequence[\lambda] == 1$ /*Si l'instant actuel porte la valeur 1.*/ **then**
      $V'_c := c_{id}$; /*On initialise V'_c à c_{id}*/
    else if $Sequence[\lambda] == 0$ /*Si l'instant actuel porte la valeur 0.*/ **then**
      $V'_c := 1$; /*On initialise V'_c à 1*/
    end if
  end for
end if

Figure 6.9: Algorithme d'ordonnancement multitâches mono-processeur.

multi-tâches/mono-processeur pour l'ensemble des processeurs. En conséquence, pour chaque processeur, on a soit un ordonnancement mono-tâche/mono-processeur ou un ordonnancement multi-tâches/mono-processeur. Ces deux cas ont été abordé précédemment.

6.4 Discussion

Nous avons présenté dans ce chapitre une méthodologie de conception d'architectures massivement parallèles et leurs associations sur des applications hautes performances. La modélisation est faite par le biais du profil Marte, dédié à la modélisation de systèmes embarqués temps réel.

Nous avons ensuite capturé le comportement d'un système, co-modélisé en Marte, par le biais d'horloges abstraites. Nous avons présenté des horloges fonctionnelles décrivant un ordre d'exécution de tâches. Nous avons aussi présenté des horloges physiques représentant les vitesses de traitements des différents processeurs. Ensuite, une projection d'horloges abstraites fonctionnelles sur des horloges abstraites physiques a été proposée. Le résultat de cette projection est une trace d'horloges abstraites simulant l'activité de processeurs durant l'exécution de fonctionnalités. Ces horloges nous seront utiles dans le chapitre 7 pour analyser de propriétés fonctionnelles (ordre d'exécution de tâches) et non fonctionnelles (temps d'exécution et consommation d'énergie) de systèmes co-modélisés en Marte.

Chapter 7

Méthodes d'analyse de propriétés de MPSoC basées sur les horloges abstraites

7.1 Présentation générale

L'exécution d'une application, dont le comportement est synchrone, sur une architecture potentiellement asynchrone pourra engendrer des violations de l'ordre d'exécution de tâches, dû aux latences de calculs et de communications. Il est donc important de vérifier, et de préserver si c'est possible, les contraintes de précédences et de périodicités de tâches après l'étape d'association d'une application avec une architecture. Typiquement, un processeur ne doit pas commencer l'exécution d'une tâche avant que les données d'entrées ne soient présentes. Ce fait est capturé et analysé sur la trace d'horloges d'exécution de type ternaire (contenant les valeurs "0", "1" et "-1"). Dans ce chapitre, une trace d'horloges ternaires est évaluée afin de choisir une configuration optimale vis-à-vis des consommations de l'énergie et du respect des contraintes temporelles.

Dans le chapitre 5, nous avons d'abord modélisé des applications hautes performances en utilisant le profil Marte. Nous avons ensuite abstrait le modèle résultant par le biais d'horloges fonctionnelles. Dans le chapitre 6, nous avons modélisé une architecture multiprocesseur et l'allocation entre une architecture et une application,

aussi via le profil Marte. Nous avons ensuite décrit le comportement temporel d'un système co-modélisé en Marte par le biais d'horloges abstraites simulant l'activité des processeurs durant l'exécution de fonctionnalités.

Dans le présent chapitre, nous proposons des méthodes d'analyses et de vérifications, à base d'horloges abstraites, de propriétés fonctionnelles et non fonctionnelles.

Dans la section 7.2, nous introduisons une méthode d'analyse *post-mortem* de propriétés et une méthode de synthèse de propriétés. Dans la section 7.3, des méthodes d'analyse de propriétés fonctionnelles sont présentées sur la partie fonctionnelle uniquement, d'une part, et sur le système complet d'autre part. Dans la section 7.4, nous abordons l'analyse de propriétés non fonctionnelles telles que l'estimation des fréquences minimales des processeurs, le calcul du temps d'exécution et la consommation d'énergie totale dissipée.

7.2 Deux méthodes de raisonnements

Nous allons introduire dans cette section deux méthodes de raisonnements de systèmes, co-modélisés en Marte, en se basant sur des horloges abstraites :

1. *analyse post-mortem de propriétés* : cette méthode consiste à analyser et à vérifier des horloges abstraites, simulant l'activité des processeurs, durant l'exécution de fonctionnalités. Le concepteur choisit les fréquences des différents processeurs, avant la phase d'analyse temporelle ;

2. *synthèse de propriétés* : cette méthode consiste à construire une trace d'horloges abstraites d'exécution "*correcte par construction*". En d'autres termes, ces horloges vérifient les contraintes temporelles imposées durant la phase de modélisation de l'application. Les fréquences des différents processeurs sont automatiquement fixées après la phase d'analyse, à l'opposé de la méthode *post-mortem*.

7.2.1 Analyse *post-mortem* de propriétés

Dans les chapitres 5 et 6, nous avons montré comment abstraire des modèles co-modélisés en Marte. Cela est fait par le biais d'horloges abstraites ayant les types suivants :

- **fonctionnelle** : une horloge abstraite fonctionnelle spécifie le nombre d'activation de tâches dans un graphe de tâches ainsi que les dépendances avec les autres tâches. Une trace d'horloges fonctionnelles décrit un ordre d'exécution de tâches ;

- **physique** : une horloge abstraite physique trace les tops d'horloge d'un processeur. Ces tops sont synchronisés sur une horloge de référence ;

- **exécution** : une horloge abstraite d'exécution simule l'activité d'un processeur durant l'exécution de fonctionnalités. Elle est le résultat d'une projection d'une ou de plusieurs horloges fonctionnelles sur une horloge physique.

Les horloges d'exécution ont pour but de simuler l'activité des processeurs durant l'exécution des tâches fonctionnelles. Cependant, des contraintes de dépendances de tâches, imposées durant la modélisation de l'application, doivent être vérifiées après la phase d'association. En effet, la correction du comportement temporel de l'application dépend de la puissance de calcul des processeurs. De plus, les applications considérées

sont en temps réel dur. Donc, le dépassement des échéances remet en cause la structure de l'application (les contraintes de précédences par exemple).

En choisissant la méthode d'analyse *post-mortem* de propriétés, nous vérifions par inspection, sur une trace d'horloges d'exécution, les contraintes temporelles suivantes :

- *précédence et périodicité :* avant de pouvoir démarrer, une tâche doit attendre le résultat d'un travail d'une autre tâche. Ce genre de contraintes est facilement observable dans une trace d'horloges fonctionnelles [1]. Nous vérifions sur la trace d'horloges d'exécution si ces contraintes sont toujours vérifiées ;
- *date d'échéance :* nous considérons des systèmes temps réel durs dans notre analyse. Pour cela, il s'agit de la date à laquelle les travaux doivent nécessairement être terminés, par rapport à leur date de création. Nous vérifions sur la trace d'horloges d'exécution si la durée d'exécution des tâches dépasse les échéances imposées par le concepteur ou par l'environnement.

Une fois les contraintes fonctionnelles vérifiées, des propriétés non fonctionnelles peuvent être aussi analysées. Par exemple, des informations concernant les durées d'exécutions de tâches ou la consommation d'énergie totale d'un ensemble de configurations possibles d'un système, peuvent réduire considérablement l'espace de solutions considérées.

Le raisonnement *post-mortem* consiste donc à analyser des propriétés fonctionnelles et non fonctionnelles sur des horloges d'exécution. Cette analyse est faite après la phase d'allocation d'applications sur des architectures.

7.2.2 Synthèse de propriétés

La synthèse de propriétés, quant à elle, consiste à fixer les valeurs de fréquences des différents processeurs impliqués dans une exécution. Ces valeurs assurent la correction de l'exécution des tâches vis-à-vis des contraintes suivantes :

- temps d'échéance : les échéances imposées sur l'exécution d'une fonctionnalité sont respectées. En effet, les fréquences des processeurs sont choisies de telles sortes que les dates d'échéances soient forcément respectées ;
- dépendances de tâches : les fréquences des différents processeurs assurent l'exécution des différentes tâches selon les contraintes de précédences et de périodicités imposées durant la modélisation des fonctionnalités.

La détermination d'une fréquence minimale d'un processeur repose sur deux conditions. La première condition est associée à la charge de travail d'un processeur et aux temps d'échéances imposés sur la terminaison des calculs. C'est une contrainte qui doit être toujours respectée, surtout dans les systèmes strictes. La deuxième contrainte est associée à l'ordre d'exécution de tâches. En effet, quand une architecture multiprocesseur est considérée, plusieurs processeurs sont actifs en même temps et exécutent différentes tâches, pouvant être dépendantes l'une de l'autre. Il est donc primordial de respecter ces dépendances.

Une fois les fréquences de processeurs fixées, une analyse de propriétés non fonctionnelles doit être considérée. Par exemple, il est intéressant de dimensionner les tailles de mémoires ainsi que d'estimer des temps d'exécution de tâches et de la consommation d'énergie totale.

[1] La sémantique de l'extraction de contraintes temporelles est définie dans le chapitre 5.

7.3 Analyse de propriétés fonctionnelles

Dans cette section, nous nous intéressons à l'analyse et la vérification de propriétés fonctionnelles d'un système co-modélisé en Marte. Nous distinguons les deux types d'analyse suivants :

- *avant allocation à partir d'un modèle d'application :* nous proposons une analyse de fonctionnalités indépendamment d'architecture. Dans cette analyse, nous nous intéressons à l'étude de synchronisabilité de tâches fonctionnelles. Cela nous permet d'évaluer la conception d'une application. Par exemple, quand deux composants sont synchronisables, des tampons de communications de tailles finies peuvent être insérés pour la mémorisation des données générées et non pas encore consommées ;
- *après allocation à partir d'un modèle logiciel/matériel associé :* nous proposons une analyse d'un système complet contenant de fonctionnalités associées à des ressources physiques. Nous vérifions si les contraintes de dépendances de tâches et les échéances sont toujours respectées après la phase d'allocation de l'application sur l'architecture. Dans cette analyse, une vérification de propriétés non fonctionnelles est aussi possible vu que des détails d'architecture sont considérés.

7.3.1 Avant allocation à partir d'un modèle d'application

Nous proposons une analyse de synchronisabilité de tâches, modélisées en Marte sous forme d'un graphe de tâches acyclique. Cette analyse consiste à vérifier si la communication inter-tâches est synchronisable. Nous nous servons des outils synchrones pour raisonner sur les tâches en les associant à des horloges abstraites.

7.3.1.1 Analyse de synchronisabilité selon la théorie des réseaux de Kahn N-synchrone

Nous souhaitons analyser le comportement temporel des composants fonctionnels de types élémentaires. Nous avons choisi les horloges abstraites fonctionnelles[2] comme support d'analyses. Ces horloges décrivent le nombre d'activation de composants élémentaires (ou tâches fonctionnelles) dans un graphe de composants, tout en conservant l'ordre d'exécution des différentes tâches. Cet ordre est représenté par des contraintes de périodicités et de précédences sur les différentes horloges fonctionnelles.

$$
\begin{array}{rl}
\mathit{IdealClk}: & 1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1\,1 \\
clk_1: & 1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0 \\
clk_2: & 0\,0\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1\,0\,1
\end{array}
$$

Figure 7.1: Trace de deux horloges binaires clk_1 et clk_2.

La Figure 7.1 montre une trace de deux horloges abstraites fonctionnelles clk_1 et clk_2. Ces deux horloges, de types périodiques, partagent une horloge de référence *IdealClk*. L'horloge clk_1 est créée par exemple en appliquant une transformation affine sur l'horloge globale définie par : $f : x \longrightarrow 2(x-1)+1$, où la valeur "2" représente la période de clk_1 et la valeur "1" représente la phase pour le début des activations. Quant à clk_2, elle est créée en appliquant une transformation affine sur l'horloge globale définie par : $g : x \longrightarrow 2(x-1)+4$, où 2 représente la période de clk_2 et la valeur "4" représente la phase. Les fonctions f et g permettent la distribution des valeurs "1" par rapport aux

[2]La sémantique des horloges fonctionnelles est définie dans le chapitre 5.

instants de l'horloge de référence. Par exemple, pour distribuer les quatre premières occurrences de la valeur "1", nous obtenons :

$$\begin{array}{l} f(1) = 1,\ f(2) = 3,\ f(3) = 5,\ f(4) = 7 \\ g(1) = 4,\ g(2){=}6,\ g(3) = 8,\ g(4) = 10 \end{array}$$

Analyse de synchronisabilité d'horloges abstraites

Afin d'analyser la synchronisabilité des horloges abstraites, nous définissons les fonctions *position*() et *taille*() dont la sémantique est empruntée de Cohen et al. [29].

Definition 11 (La fonction *position()*) *Étant donné une horloge fonctionnelle clk, la fonction position*(n, clk) *dénote la position de la* $n^{ième}$ *occurrence d'un instant portant la valeur "1" sur l'horloge clk. Nous écrivons alors,* $position(n, clk) = q$ *où* $n, q \in \mathbb{N}^*$.

Prenons l'exemple de l'horloge binaire clk_1. Les deux fonctions, $position(2, clk_1) = 3$ et $position(4, clk) = 7$, calculent respectivement la position de la deuxième et de la quatrième occurrence d'un instant portant la valeur "1" sur l'horloge clk_1 (la position de la première valeur binaire de l'horloge étant 1 et non pas 0).

Definition 12 (La fonction *taille()*) *Étant donné une horloge fonctionnelle clk, la fonction taille(clk) retourne le nombre de valeurs binaires "0" et "1" contenues dans clk.*

Par exemple, la longueur de l'horloge clk_1 est $taille(clk_1) = 24$, où 24 représente le nombre total d'occurrences des valeurs "0" et "1" dans clk_1.

Nous analysons maintenant les positions des instants portants la valeur "1" sur des couples d'horloges abstraites (clk_i,clk_j) afin d'étudier la synchronisabilité des ces deux horloges. Deux cas se présentent :

1. **horloges finies** : quand clk_i et clk_j ont une taille finie, il est possible de synchroniser les communications en insérant des mémoires tampons de tailles finies.

2. **horloges infinies** : quand clk_i et clk_j ont des comportements sporadiques, il est impossible d'analyser la synchronisabilité de ces deux horloges. Cependant, quand elles ont un comportement périodique, deux cas se présentent :

   - si nous pouvons trouver des périodes pour clk_i et clk_j contenant le même nombre de "1", alors il est possible de synchroniser la communication entre clk_i et clk_j ;
   - sinon, il est impossible de synchroniser la communication entre clk_i et clk_j.

Prenons comme exemple un composant C_1 générateur de données, ayant une horloge fonctionnelle clk_1, et un autre composant C_2 consommateur de données, ayant une horloge fonctionnelle clk_2. Quand clk_1 et clk_2 ont des tailles finies, la communication entre les C_1 et C_2 peut être synchronisée : les données générées par C_1 et non pas encore consommées C_2 sont stockées dans des mémoires tampons de taille bornée. Cependant, quand clk_1 et clk_2 sont infinies, les données générées par C_1 doivent être consommées entièrement par C_2 (même nombre de "1" dans les périodes de clk_1 et clk_2). Quand cela est le cas, une étape de synchronisation de la communication est faite, entre C_1 et C_2, en insérant une mémoire tampon de taille bornée. Quand les données générées par C_1 sont partiellement consommées par C_2 au cours du temps (différents nombres de 1 dans les périodes de clk_1 et clk_2), il est impossible de synchroniser la communication avec des mémoires tampon de taille bornée.

Soit nb_1 le nombre total d'occurrence de la valeur "1" dans clk_1
Soit nb_1 le nombre total d'occurrence de la valeur "1" dans clk_2
$\forall\ n \in \mathbb{N}$
if $position(n,clk_1) = position(n,clk_2)$ **then**
Le canal Ch_1 qui relie P_{Out} de C_1 à P_{In} de C_2 n'a pas besoin d'une mémoire tampon ou d'un système de ralentissement, car la communication entre ces deux ports est synchrone.
else if $position(n,clk_1) < position(n,clk_2)$ **then**
Le nombre d'activations de C_1 est le même que C_2 mais avec un certain retard d. Cela signifie que C_1 produit des données et C_2 consomme la même quantité de données, mais après d instants logiques. Par conséquent, des mesures devraient être prises pour permettre la synchronisation des deux horloges afin d'éviter la perte de données (voir les sections suivantes).
else if $position(n,clk_1) > position(n,clk_2)$ **then**
C_2 consomme des données avant qu'elles ne soient produites par C_1, ce qui rend le système incohérent car les données ne sont pas présentes lors de l'activation de C_2. Pour cela, des mesures devraient être prises pour ralentir clk_1 ou pour accélérer clk_2 dans le but de satisfaire les exigences du système.
end if

Figure 7.2: Algorithme d'analyse de synchronisabilité des horloges clk_1 et clk_2.

La Figure 7.2 illustre un algorithme d'analyse de synchronisabilité entre deux horloges clk_1 et clk_2, ayant des tailles finies. Ces horloges sont associées respectivement aux composant producteur C_1 et au composant consommateur C_2. Nous présentons trois cas de communication possibles :

1. la communication entre le producteur et le consommateur est synchrone. Cette communication est considérée idéale et aucune intervention n'a lieu ;
2. le producteur a un rythme de production plus rapide que celui du consommateur ;
3. le rythme de producteur est plus lent que celui du consommateur.

Calcul d'une taille minimale de mémoire tampon

Nous calculons maintenant la taille minimale de mémoire tampon nécessaire pour la synchronisation d'une communication entre deux tâches.

Definition 13 (La fonction *Quantité*()) *Soit clk une horloge binaire. Les expressions* $(1.clk, i)$ *et* $(0.clk, i)$ *expriment respectivement que la* $i^{ième}$ *position dans l'horloge clk est* 1 *et* 0, $\forall i \in [1, taille(clk)]$. *La fonction Quantité est alors définie récursivement comme suit :*

- $Quantité(1.clk, i) = 1 + Quantité(clk, i-1)$
- $Quantité(0.clk, i) = 0 + Quantité(clk, i-1)$
- $Quantité(1.clk, 1) = 1$
- $Quantité(0.clk, 1) = 0$

En appliquant la fonction *Quantité*(clk, i) sur les horloges clk_1 et clk_2 de la Figure 7.1, nous obtenons le nombre d'activations de composants jusqu'à l'instant logique i :

Quantité(clk_1,1)=1, *Quantité*(clk_2,1)=0
Quantité(clk_1,2)=1, *Quantité*(clk_2,2)=0
Quantité(clk_1,3)=2, *Quantité*(clk_2,3)=0
Quantité(clk_1,4)=2, *Quantité*(clk_2,4)=1
Quantité(clk_1,5)=3, *Quantité*(clk_2,5)=1
Quantité(clk_1,6)=3, *Quantité*(clk_2,6)=2
...

Grâce à la fonction *Quantité*(), nous sommes en mesure de calculer la quantité de données (représentée par des occurrences de valeurs "1") générées par un producteur et non pas encore consommées. Nous calculons ainsi la différence pour chaque instant logique de clk_1 et clk_2, entre le nombre de 1 produit et le nombre de 1 consommé.

Definition 14 (La fonction *Différence*()) *Soient clk_x, $clk_y \in Clk$. La fonction Différence() est définie comme suit : Différence(clk_x,clk_y,i) = $|$Quantité(clk_x,i) - Quantité(clk_y,i)$|$ $\forall$ i $\in$ [1, taille(clk_x)].*

En appliquant la fonction *Différence*() sur les différents instants logiques des horloges clk_1 et clk_2, nous pouvons déterminer le décalage des positions de la valeur "1" entre clk_1 et clk_2, à chaque instant logique :

Différence(clk_1,clk_2,1)=1
Différence(clk_1,clk_2,2)=1
Différence(clk_1,clk_2,3)=2
Différence(clk_1,clk_2,4)=1
Différence(clk_1,clk_2,5)=2
Différence(clk_1,clk_2,6)=1
...

Puisque *Différence*(clk_1,clk_2,5)=2, le producteur associé à l'horloge clk_1 produit, au cinquième instant, deux unités de données avant que le consommateur, associé à clk_2, consomme. Nous pouvons ainsi remarquer un problème de synchronisation entre les horloges clk_1 et clk_2 vu que les instants logiques des deux horloges ne sont pas superposables. Le délai de communication entre des horloges *mal ajustées* est synchronisé en insérant des mémoires tampon.

Allocation de mémoires tampons

Le concepteur peut envisager un mécanisme de communication entre composants, contenant une mémoire tampon partagée, permettant de mémoriser temporairement les données échangées entre des composants. L'insertion de mémoires tampons doit être limitée à l'endroit où cela est nécessaire pour économiser le coût de conception de puces et de leurs tailles physiques. Notre approche permet d'identifier les endroits qui nécessitent une insertion de mémoire, mais cela n'est pas suffisant. En effet, il faut pouvoir estimer une taille minimale de mémoires tampons à utiliser pour obtenir une configuration optimale.

Dans la Figure 7.1, nous pouvons remarquer un problème de synchronisation entre les deux horloges clk_1 et clk_2 dû à la valeur de l'offset de clk_2.Par conséquent, la communication n'est pas instantanée. Pour stocker les données produites et pas encore

consommées, nous déterminons la taille minimale d'une mémoire tampon qui permet le stockage de données sans perte d'information.

Pour cela nous définissons la fonction *TailleMinimale*() suivante :

Definition 15 (Taille minimale) $\forall$ *clk_x, $clk_y \in Clk$ et $\forall$ $i \in$ [1,taille(clk_x)], la taille minimale de la mémoire tampon entre les horloges clk_x et clk_y est définie par TailleMinimale(clk_x,clk_y) = Maximum(Différence(clk_x,clk_y,i)).*

En appliquant la fonction *TailleMinimale* sur les horloges clk_1 et clk_2, nous déterminons une mémoire tampon de taille minimale égale à 2. L'unité de mesure de la taille de cette mémoire est strictement liée à la nature et à la taille des données traitées.

7.3.1.2 Analyse de synchronisabilité en Signal

Dans cette section, nous abordons l'analyse de synchronisabilité d'horloges ayant des contraintes de périodicités. Nous considérons un codage de contraintes d'horloges fonctionnelles en utilisant le langage synchrone Signal. Le codage est fait d'une manière globale en définissant des valeurs génériques d'offsets et de périodes. Ces valeurs génériques seront remplacées par des valeurs réelles durant l'exécution du programme final, codé en Signal. Ce dernier offre des outils très appropriés pour l'analyse de contraintes temporelles contenues dans un système.

Étude de synchronisabilité de composants

Nous considérons dans cette analyse trois horloges clk_1, clk_2 et clk_3, respectivement associées à des composants C_1, C_2 et C_3. Nous supposons l'existence, d'une part, d'une contrainte de périodicité P_1 entre C_1 et C_2, et d'autre part, une contrainte de périodicité P_2 entre C_2 et C_3. Nous spécifions les relations d'affinités entre les couples d'horloges (clk_1, clk_2) et (clk_2, clk_3) comme suit :

$$(P_1) : clk_1 \overset{(1,\phi_1,d_1)}{\rightarrow} clk_2 \tag{7.1}$$

et

$$(P_2) : clk_2 \overset{(1,\phi_2,d_2)}{\rightarrow} clk_3 \tag{7.2}$$

Selon les travaux de Smarandache et al. [111], de nombreuses opérations peuvent être définies sur les relations d'horloges affines. Parmi elles, nous citons l'opération de composition, noté $*$. En règle générale, les deux relations ci-dessus peuvent être composées comme suit :

$$clk_1 \overset{(1,\phi_1,d_1)}{\rightarrow} clk_2 \ * \ clk_2 \overset{(1,\phi_2,d_2)}{\rightarrow} clk_3. \tag{7.3}$$

Le résultat de cette composition donne une relation affine directe entre clk_1 et clk_3. Il a été démontré que cette relation est équivalente à :

$$clk_1 \overset{(1,\phi_1+d_1\phi_2,d_1d_2)}{\rightarrow} clk_3. \tag{7.4}$$

Les relations d'horloges ci-dessous caractérisent formellement le comportement temporel dans les cas des contraintes périodiques imposées dans une spécification fonctionnelle Marte. Cette description peut être utilisée pour raisonner sur certaines exigences imposées par l'environnement sur le système comme illustré ci-dessous. Par exemple, dans le cas de traitement d'image vidéo, une contrainte de qualité de service

(QoS–*Quality of Service*) est imposée sur le fonctionnement d'un système. Par exemple, nous pouvons imaginer une contrainte sur le nombre d'images par seconde que doit recevoir l'utilisateur en cas d'un algorithme de traitements d'images. Dans ce contexte, nous considérons ce critère de qualité de service qui est associé au composant producteur de données (étant associé à l'horloge clk_1 par exemple) et le composant responsable de l'affichage sur un écran (étant associé à une horloge clk_4 par exemple). Nous définissons ainsi une troisième contrainte P_3 de périodicité entre les horloges clk_1 et clk_4 :

$$(P_3) : clk_1 \overset{(1,\phi_3,d_3)}{\rightarrow} clk_4. \tag{7.5}$$

Nous pouvons maintenant utiliser les techniques d'analyses, appliquées sur des horloges affines, afin de vérifier la compatibilité des contraintes de qualité de service (P_3) avec les contraintes imposées sur le fonctionnement interne du système (P_1 et P_2). Nous effectuons donc une étude de synchronisabilité entre les horloges clk_3 et clk_4.

Selon les travaux de Smarandache et al. [111], si les deux horloges clk_3 et clk_4 sont synchronisables, elles doivent vérifier la contrainte suivante :

$$\phi_1 + d_1 \times \phi_2 = \phi_3 \tag{7.6}$$

et

$$d_1 \times d_2 = d_3. \tag{7.7}$$

La Figure 7.3 montre un programme Signal qui décrit les relations d'horloges décrites ci-dessus : P_1, P_2 et P_3. Le programme appelé `Synchronization`, prend en entrée une horloge clk_1 qui représente le taux de production de pixel (ligne **3**), et rend en sortie les les horloge clk_2, clk_3 et clk_4 (ligne **4**). Les paramètres des relations affines identifiées sont spécifiés comme des paramètres statiques d'un programme Signal (ligne **2**).

```
1   process Synchronization =
2    { integer d1, phi1, d2, phi2, d3, phi3;}
3    (? integer clk_1;
4     ! integer clk_2,clk_3,clk_4; )
5    (| clk_2 := affine_sample{phi1,d1}(clk_1)
6     | clk_3 := affine_sample{phi2,d2}(clk_2)
7     | clk_4 := affine_sample{phi3,d3}(clk_1)
8     |)
9    where
10    process affine_sample =
11     { integer phi, d; }
12     ( ? x;
13       ! y; )
14     (| v ^= x
15      | v:= (d-1) when(zv=0) default (zv-1)
16      | zv := v init phi
17      | y := x when (zv=0)
18      |)
19    where
20      integer v, zv;
21    end;%process affine_sample%
22  end;%process Synchronization
```

Figure 7.3: Spécification des relations d'horloges affines en Signal.

La relation affine est codée par le processus `affine_sample` défini à la ligne **10**. Elle prend une horloge `x` en entrée et produit une horloge `y` en sortie selon une transformation

affine ayant des paramètres `phi` et `d`. Pour plus de détails sur ce processus, le lecteur peut se référer à [16].

Afin d'analyser les contraintes de synchronisation avec le compilateur Signal, nous devrons spécifier explicitement les contraintes temporelles dues aux dépendances entre tâches et les insérer dans le processus `affine_sample` (cf. Figure 7.3). Les équations illustrées dans la Figure 7.4, codent ces contraintes en Signal. Le code doit être inséré dans le programme `Synchronization` après la ligne **7**.

```
1    (| C_clk4 := ^clk4 default not(clk4 ^+ clk3)
2     | C_clk3 := ^clk3 default not(clk4 ^+ clk3)
3     | assert(C_clk4=C_clk3) when(^C_clk4)
4        when(phi3=phi1+d1*phi2) when(d3=d1*d2)
5     |)
```

Figure 7.4: Spécification des contraintes de synchronisation d'horloges.

En modifiant le programme de la Figure 7.3 et en ajoutant les contraintes temporelles illustrées dans la Figure 7.4, nous pouvons maintenant compiler le programme final afin de vérifier formellement d'une part les compatibilités des choix de conception, et d'autre part, les exigences de qualité de service QoS selon des valeurs particulières des paramètres des relations affines. D'autre types d'analyses temporelles en Signal, comme dans Gamatié et al. [51], peuvent être envisagés.

7.3.2 Après allocation à partir d'un modèle logiciel/matériel associé

Nous avons vu dans le chapitre 6, comment obtenir des horloges abstraites d'exécution à partir d'un modèle Marte. Nous proposons dans ce qui suit deux méthodes d'analyse et de vérification d'horloges d'exécution. La première analyse, appelée *post-mortem*, permet de vérifier la correction d'une trace d'horloges d'exécution vis-à-vis de l'ordre d'exécution de tâches. Cet ordre est imposé durant la modélisation de fonctionnalités, sous forme de contraintes de précédences et de périodicités appliquées sur des tâches communicantes. La deuxième analyse est une synthèse de propriétés. Elle consiste à abstraire le système par des horloges d'exécution respectant, par construction, l'ordre d'exécution de tâches.

7.3.2.1 Analyse *post-mortem* de propriétés

Nous avons présenté dans le chapitre 6 un algorithme d'ordonnancement de tâches sur des processeurs. Cet algorithme manipule des horloges abstraites. Il définit des projections d'horloges fonctionnelles sur des horloges physiques. Le résultat de telles projections est une trace d'horloges d'exécution.

La Figure 7.5 montre la trace de trois horloges fonctionnelles clk_1, clk_2 et clk_3. Nous considérons aussi trois processeurs P_1, P_2 et P_3, ayant respectivement les fréquences 30, 45 et 15 MHz. Ils sont associés respectivement aux horloges physiques $PhysicalClk_1$, $PhysicalClk_2$ et $PhysicalClk_3$. La Figure 7.6 (a) montre la trace de trois horloges d'exécution $TClk_1$, $TClk_2$ et $TClk_3$. Ces dernières sont le résultat du placement[3] de clk_1, clk_2 et clk_3 respectivement sur $PhysicalClk_1$, $PhysicalClk_2$ et $PhysicalClk_3$.

Cependant, les contraintes de dépendance de tâches illustrées dans la Figure 7.5 doivent être vérifiées sur la nouvelle trace d'horloges abstraites de la Figure 7.6 (a). Plus particulièrement, les contraintes suivantes sont à vérifier :

[3]La projection des horloges fonctionnelles clk_1, clk_2 et clk_3 sur les horloges physiques $PhysicalClk_1$, $PhysicalClk_2$ et $PhysicalClk_3$ a été étudiée dans le chapitre 6.

$IdealClk$: 1 1 1 1 1 1
clk_1: 1 0 0 1 0 0
clk_2: 0 1 0 0 1 0
clk_3: 0 0 1 0 0 1

Figure 7.5: Une trace de trois horloges fonctionnelles clk_1, clk_2 et clk_3.

- *contrainte 1 :* la première occurrence de la valeur "1" dans l'horloge clk_1 précède la première occurrence de la valeur "1" dans l'horloge clk_2 ;
- *contrainte 2 :* la deuxième occurrence de la valeur "1" dans l'horloge clk_1 précède la deuxième occurrence de la valeur "1" dans l'horloge clk_2 ;
- *contrainte 3 :* la première occurrence d'une valeur "1" de l'horloge clk_2 précède la première occurrence de la valeur "1" dans l'horloge clk_3 ;
- *contrainte 4 :* la deuxième occurrence d'une valeur "1" de l'horloge clk_2 précède la deuxième occurrence de la valeur "1" dans l'horloge clk_3.

En analysant la trace d'horloges de la Figure 7.6 (a), nous remarquons que la contrainte temporelle *contrainte 2* entre les horloges $Tclk_1$ et $Tclk_2$ est violée. En effet, la deuxième occurrence de la valeur "1" dans l'horloge $Tclk_2$ (position 11 sur $IdealClk$) a lieu avant que la deuxième occurrence de la valeur "1" dans l'horloge $Tclk_1$ n'ait lieu (position 19 sur $IdealClk$). Cela implique que le processeur $PhysicalClk_2$ a déclenché un travail dépendant d'un résultat indisponible à ce moment.

Afin de résoudre le problème de violation des contraintes de précédences, nous proposons deux solutions :

1. la première solution consiste à retarder l'activation de l'horloge la plus rapide qui cause le problème de violation de contraintes. Ce retard ou ajournement d'activation peut être vu physiquement par un contrôle sur l'initialisation d'une tâche par un processeur.
2. la deuxième solution consiste à modifier le choix des fréquences associées aux différents processeurs impliqués dans l'exécution. La modification de la fréquence d'un processeur induit une altération de la distribution des instants de l'horloge fonctionnelle sur l'horloge d'exécution. Ainsi, en choisissant une fréquence adéquate, les contraintes de précédences seront de nouveaux validées avec la nouvelle configuration des processeurs.

Les deux solutions ci-dessus visent à réorganiser la distribution des valeurs ternaires 0, 1 et -1 sur les horloges $Tclk_1$, $Tclk_2$ ou $Tclk_3$ jusqu'à l'obtention d'une nouvelle trace qui vérifie les contraintes temporelles et les contraintes de dépendance de tâches.

Ajournement de l'activation d'un processeur

Une modélisation fonctionnelle d'une application peut être réalisée de deux manières : en horizon fini ou infini. Cela revient à exécuter l'application en un temps fini ou infini. L'ajournement de l'activation d'un processeur consiste à retarder le début d'exécution de l'horloge la plus rapide qui cause le problème de violation de contrainte. Cette solution est applicable seulement sur une trace d'horloges abstraites finies.

La Figure 7.6 (b) montre une nouvelle trace des horloges d'exécution Tlk_1, Tlk_2 et Tlk_3 avec un ajournement des activations du processeur associé à l'horloge $Tclk_2$. Cet

Figure 7.6: Légende: (a) trace de simulation des trois horloges abstraites $Tclk_1$, $Tclk_2$ et $Tclk_3$ et (b), ajournement de l'activation de l'horloge $Tclk_2$ en insérant quatre instants "idle", représentés par quatre valeurs de -1, et en décalant vers la droite les autres instants logiques.

ajournement est représenté sur l'horloge $Tclk_2$ par l'insertion d'une séquence de quatre instants logiques portant la valeur "−1". Cette insertion est faite à partir du premier instant logique. Cela induit naturellement un décalage vers la droite des autres instants logiques de l'horloge $Tclk_2$. Nous remarquons dans cette nouvelle trace que toutes les contraintes de précédences entre les couples d'horloges ($Tclk_1$,$Tclk_2$) et ($Tclk_2$,$Tclk_3$) sont respectées.

Modification de la fréquence de processeurs

Considérons toujours l'exemple de la Figure 7.6 (a). Afin de régler le problème de violation des contraintes temporelles, nous diminuons maintenant la fréquence du processeur `Proc2` de 45 MHz à 15 MHz. Cette modification engendre une nouvelle trace de l'horloge $PhysicalClk_2$ dont les instants sont étirés plus vers la droite.

Nous projetons, en appliquant l'algorithme d'ordonnancement mono-tâche/mono-processeur, les horloges clk_1, clk_2 et clk_3 sur la nouvelle trace des horloges physiques $PhysicalClk_1$, $PhysicalClk_2$ et $PhysicalClk_3$. Le résultat de ces projections est une nouvelle trace d'horloges d'exécutions.

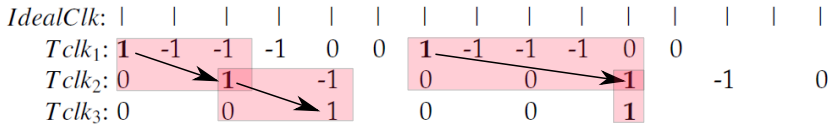


Figure 7.7: Nouvelle trace des horloges ternaires $Tclk_1$, $Tclk_2$ et $Tclk_3$ pour des fréquences `f1`=30 MHz, `f2`=15 MHz, `f3`=15 MHz.

La Figure 7.7 montre la nouvelle trace des trois horloges ternaires $Tclk_1$, $Tclk_2$ et $Tclk_3$. Ces horloges représentent l'activité des processeurs `Proc1`, `Proc2` et `Proc3` durant l'exécution de fonctionnalités et ayant les fréquences respectives 30, 15 et 15 MHz. En analysant l'occurrence des valeurs "1" sur les horloges $Tclk_1$ et $Tclk_2$, nous remarquons que les contraintes de précédences *contrainte 2* et *contrainte 3*, imposées durant la modélisation de l'application, sont respectées dans cette nouvelle distribution :

- *contrainte 1 :* la première occurrence de la valeur "1" dans l'horloge $Tclk_1$ (position 1 selon les instants de *IdealClk*) précède l'occurrence de la première valeur "1" dans l'horloge $Tclk_2$ (position 3 selon les instants de *IdealClk*) ;

- *contrainte 2 :* la deuxième occurrence de la valeur "1" dans l'horloge $Tclk_1$ (position 7 selon les instants de *IdealClk*) précède la deuxième occurrence de la valeur "1" dans l'horloge $Tclk_2$ (position 11 selon les instants de *IdealClk*).

Les contraintes *contrainte 3* et *contrainte 4* sont aussi valides dans cette nouvelle trace.

Deux avantages peuvent être tirés de cette nouvelle configuration :

1. diminution de la consommation de l'énergie dynamique due à la diminution de la fréquence du processeur `Proc2` ;

2. correction des contraintes de dépendance de tâches imposées durant la phase de modélisation de l'application.

De la discussion des deux solutions proposées ci-dessus, nous observons que les horloges abstraites offrent une vue expressive, à un haut niveau d'abstraction, du comportement d'un système co-modélisé en Marte.

7.3.2.2 Synthèse de propriétés

Dans cette section, nous proposons un ordonnancement, de type statique non préemptif, de tâches fonctionnelles sur de processeurs. Quand appliquer sur un modèle Marte, cet algorithme assure l'ordre d'exécution de tâches fonctionnelles ainsi que les échéances imposées sur la terminaison de fonctionnalités. Tout cela est fait par le biais d'horloges abstraites.

La méthode d'analyse consiste à tracer l'activité des différents processeurs en terme de cycles d'horloge. Cela nous permet de déterminer des ratios de fréquences entre les différents processeurs communicants. Les ratios, une fois considérés pour les fréquences des différents processeurs d'un système, préservent l'ordre d'exécution de tâches.

Projection d'une horloge fonctionnelle étendue sur une horloge physique

Dans le cas d'un ordonnancement mono-tâche/mono-processeur, une projection d'une horloge fonctionnelle sur une horloge physique est considérée. Afin d'y arriver, deux étapes doivent être suivies :

- expansion de l'horloge fonctionnelle en considérant le nombre de cycles processeur pour l'exécution d'une instance de tâche ;
- projection de l'horloge étendue sur une horloge physique ;

Expansion d'horloges fonctionnelles : nous considérons une horloge abstraite fonctionnelle clk_i décrivant le nombre d'activation d'une tâche T_i. L'horloge clk_i doit être d'abord étendue avant d'être allouée sur une horloge physique. L'expansion comporte des informations relatives à l'exécution de la tâche T_i. Plus concrètement, chaque instant de clk_i portant la valeur "1" est remplacé par une séquence s comportant c_i valeurs. La séquence s commence par une valeur "1" suivie de $(c_i - 1)$ valeurs de "-1". La séquence s représente le nombre de cycles d'horloge processeur nécessaire pour exécuter une répétition de la tâche T_i. Nous notons clk'_i l'horloge résultante de cette expansion. Le nombre total d'instant dans clk'_i représente le nombre de cycles processeurs nécessaire pour terminer l'exécution de toute les répétitions de la tâche T_i.

clk_1:	1 0 0 0 0 1 0 0 0	clk'_1:	1 -1 -1 0 0 0 0 1 -1 -1 0 0 0
clk_2:	0 1 0 0 0 0 1 0 0	clk'_2:	0 1 -1 -1 -1 0 0 0 0 1 -1 -1 -1 0 0
	(a)		(b)

Figure 7.8: Légende: (a) une trace de deux horloges abstraites fonctionnelles finies clk_1 et clk_2 et (b), une trace de deux horloges clk'_1 et clk'_2 résultantes de l'expansion de clk_1 et clk_2 avec $c_1 = 3$ et $c_2 = 4$.

La Figure 7.8 (a) est un exemple de deux horloges fonctionnelles clk_1 et clk_2 portant des valeurs binaires "0" et "1". La Figure 7.8 (b) montre le résultat de l'expansion de clk_1 et clk_2. Les instants des nouvelles horloges clk'_1 et clk'_2 portent des valeurs de type ternaires (0, 1 ou -1). La valeur $c_1 = 3$ (resp. $c_2 = 4$) représente le nombre de cycles processeur nécessaire pour exécuter une instance de la tâche associée à clk_1 (resp. clk_2).

Projection d'une horloge étendue sur une horloge physique : nous définissons maintenant l'allocation d'une horloge étendue clk'_i sur une horloge physique *PhysicalClk* selon une bijection. En supposant que le nombre d'instants de clk'_i (soit sa longueur) est inférieure ou égale à celui de *PhysicalClk*, pour chaque entier $k \in [1, taille(clk'_i)]$, l'état

0, 1 ou −1 du $k^{\text{ième}}$ instant de clk'_i est alloué au $k^{\text{ième}}$ instant de *PhysicalClk*. Pour les $k^{\text{ième}}$ instants de *PhysicalClk*, tel que $k > taille(clk'_i)$, Ils portent la valeur "0".

Projection de plusieurs horloges fonctionnelles étendues sur une horloge physique

Dans la solution envisagée, un découpage en tranche est appliqué sur les horloges abstraites fonctionnelles. Prenons le cas d'un processeur *Proc* ayant une horloge physique *PhysicalClk* et n horloges fonctionnelles. Chaque horloge fonctionnelle clk_i est associée à une tâche fonctionnelle T_i, où $i \in [1..n]$.

Nous partitionnons les horloges clk_i en petites tranches T_i^j de même taille t_i. Une tranche T_i^j peut être vue comme étant une sous horloge fonctionnelle de l'horloge clk_i.

Exemple de partitionnement d'une horloge fonctionnelle : étant donné une horloge binaire $clk_i = 10001000$ et étant donné $t_i = 2$, l'horloge clk_i est divisée en quatre tranche comme suit : $T_i^1 = 10$, $T_i^2 = 00$, $T_i^3 = 10$ et $T_i^4 = 00$. Les différentes tranches sont étendues, avec $c_i = 2$, afin de les allouées sur *PhysicalClk*. Nous notons $T_i'^j$ les nouvelles tranches étendues : $T_i'^1 = 1-1\,0$, $T_i'^2 = 00$, $T_i'^3 = 1-1\,0$ et $T_i'^4 = 00$.

Après l'obtention des tranches $T_i'^j$ étendues des différentes horloges clk_i, nous projetons les instants de $T_i'^j$ sur l'horloge *PhysicalClk*. Cette projection représente l'exécution du processeur *Proc* des différentes tâches associées aux tranches $T_i'^j$. *Proc* commute en plusieurs passes l'exécution des différentes tranches selon l'ordre suivant :

- $1^{\text{ière}}$ passe : exécution par ordre les tranches $T_1^1, T_2^1, ..., T_n^1$;
- $1^{\text{ième}}$ passe : exécution par ordre les tranches $T_1^2, T_2^2, ..., T_n^2$;
- ...
- $p^{\text{ième}}$ passe : exécution par ordre les tranches restantes, où p représente le plus grand nombre de tranches dans les horloges clk_i.

Exemple : Nous considérons un exemple de deux horloges fonctionnelles clk_i et clk_j représentant respectivement l'activation des tâches T_i et T_j. Ces deux horloges sont exécutées sur le processeur *Proc* ayant une horloge physique *PhysicalClk*. Les tranches étendues de clk_i et clk_j vont être placées sur *PhysicalClk*. Un scénario d'ordonnancement possible serait : $T_i'^1, T_j'^1, T_i'^2, T_j'^2, T_i'^3, T_j'^3, T_i'^4, T_j'^4$. Ici, le processeur exécute d'abord la première tranche étendue $T_i'^1$ et commute ensuite le contexte vers l'exécution de la première tranche étendue $T_j'^1$, et ainsi de suite jusqu'à l'exécution de toute les tranches étendues des horloges clk_i et clk_j.

Détermination des ratios de fréquences

Les dépendances de données, explicitement spécifiées dans une trace d'horloges fonctionnelles, peuvent être violées après la phase d'expansion d'horloges. Prenons comme exemple deux horloges clk_i et clk_j associées respectivement aux tâches T_i et T_i. La trace des deux horloges clk_i et clk_j est illustrée dans la Figure 7.9 (a).

Nous déduisons de cette spécification les contraintes de dépendances de tâches suivantes :

- *contrainte*1 : les deux premières activations de la tâche T_i précèdent la première activation de la tâche T_j ;

- *contrainte*2 : la troisième et la quatrième activation de la tâche T_i précèdent la deuxième activation de la tâche T_j.

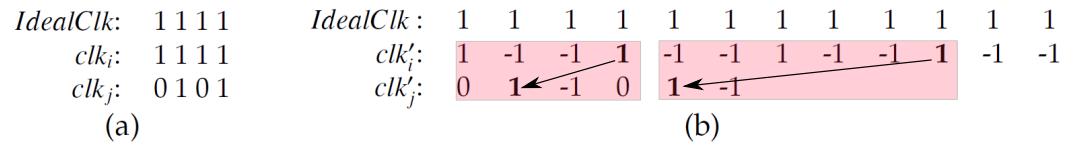


Figure 7.9: Légende: (a) une trace de deux horloges binaires finies clk_i et clk_j et (b), une trace de deux horloges étendues clk'_i et clk'_j résultantes de l'expansion de clk_i et clk_j avec $c_i = 3$ et $c_j = 2$ et ayant les contraintes de dépendances violées.

Nous étendons les deux horloges clk_i et clk_j de telle sorte que les contraintes *contrainte 1* et *contrainte 2* soient violées. Pour cela, nous considérons les cycles d'horloges $c_i = 3$ et $c_j = 2$, associées respectivement aux tâches T_i et T_j. La Figure 7.9 (b) montre une trace de deux horloges étendues clk'_i et et clk'_j. Ces horloges sont le résultat de cette expansion. Nous remarquons que les contraintes de dépendances *contraintes*1 et *contrainte*2 sont violées :

- **violation de la contrainte 1** : la première activation de l'instance T_j s'est produite à deux instants logiques avant la seconde activation de la tâche T_i ;

- **violation de la contrainte 2** : la deuxième activation de l'instance T_j s'est produite à cinq instants logiques avant la quatrième activation de la tâche T_i.

Afin de résoudre la violation des deux contraintes précédentes, une solution possible est de diminuer la fréquence du processeur le plus rapide $Proc_i$ ou d'élever la fréquence de $Proc_j$, le plus lent. Une autre solution serait de retarder l'activation du processeur le plus rapide jusqu'à ce que les contraintes soient respectées de nouveau.

Definition 16 (Ratio de fréquences) *Étant donné deux horloges étendues clk'_i et clk'_j associées respectivement à deux processeurs $Proc_i$ et $Proc_j$, le ratio de fréquences entre $Proc_i$ et $Proc_j$ est le rapport entre la fréquence f_i du processeur $Porc_i$ et la fréquence f_j du processeur $Proc_j$. Ce rapport est déduit en analysant les contraintes de dépendances de tâches entre clk'_i et clk'_j.*

Dans le but d'estimer le ratio de fréquences entre deux horloges abstraites étendues, nous analysons les grandes périodes des deux horloges en questions.

Definition 17 (Grande période) *Soit deux horloges fonctionnelles clk_i et clk_j ayant une contrainte temporelle périodique comme suit : $clk_i(n) = \alpha clk_j(n) + \beta$ et clk'_i et clk'_j leurs extensions. Soit inb_i (resp. inb_j) le nombre d'instant entre $(\alpha + 1)$ occurrences de 1 dans clk'_i (deux occurrences de la valeur "1" dans clk'_j). Les grandes périodes GP_i de clk_i et GP_j de clk_j sont définies comme suit :*

$$GP_i = inb_i + 1 \quad et \quad GP_j = inb_j + 1 \tag{7.8}$$

Dans le cas des deux horloges clk'_i et clk'_j de l'exemple précédent, une contrainte de périodicité entre ces deux horloges dicte que $clk_i = 2 \times clk_j$. Pour cela $GP_i = 6$ et $GP_j = 3$. La valeur $\frac{GP_i}{GP_j}$ représente le *ratio de fréquence* du couple (f_i, f_j), où f_i et f_j sont respectivement les fréquences des processeurs $Proc_i$ et $Proc_j$. Cela nous amène à l'équation suivante : $f_i = 2 \times f_j$.

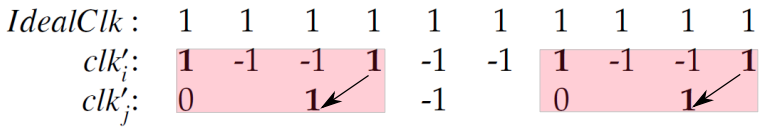


Figure 7.10: Une nouvelle trace des deux horloges clk'_i et clk'_j, après la prise en considération du ratio des fréquences respectives f_i et f_j.

Afin de régler les violations de contraintes d'horloges, nous considérons maintenant les ratios de fréquences du couple (f_1, f_2). La Figure 7.10 montre la nouvelle trace des horloges ternaires clk'_i et clk'_j en prenant en compte le *ratio de frgéquence* entre f_i et f_j. Dans cette Figure, la violation de la contrainte 1 est toujours présente : la première occurrence de la valeur "1" dans l'horloge clk'_j précède d'un instant logique l'occurrence de la deuxième occurrence de la valeur "1" dans l'horloge clk'_i. Cependant, les grandes périodes des deux horloges clk'_i et clk'_j sont désormais synchronisées au rythme de la contrainte de périodicité. En effet, la deuxième occurrence de la valeur "1" dans l'horloge clk'_j précède également d'un instant logique l'occurrence de la quatrième occurrence de la valeur "1" dans clk'_i. Cela indique que les deux processeurs ont des vitesses d'exécution cohérentes avec les contraintes imposées.

En choisissant n'importe quelle fréquence f_i du processeur $Proc_i$, si le processeur $Proc_j$ a une valeur de fréquence f_j deux fois moins rapide que celle de f_i, alors la communication entre les deux processeurs est synchronisée.

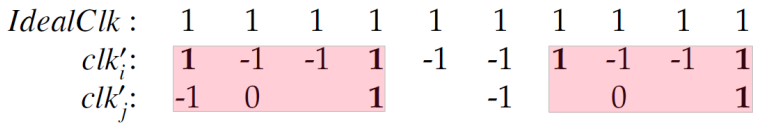


Figure 7.11: Retardement de clk'_j d'un instant logique par l'insertion d'un instant portant la valeur "-1".

Afin de régler la violation de la contrainte 1, nous ajoutons un retardement d'un instant logique du début d'activation de l'horloge clk'_j. La Figure 7.11 montre une trace des horloges clk'_i et clk'_j ayant les contraintes temporelles respectées. Le retardement est représenté par l'insertion d'une valeur -1 au début de l'horloge clk'_j. Dans un système électronique, retarder l'activation d'un processeur revient en général à réduire significativement la tension fournit au processeur. Cette technique est appliquée afin d'éviter tout accès inutile à la mémoire en cas d'absence de données (cash miss).

7.4 Analyse de propriétés non fonctionnelles

Dans cette section, nous analysons des propriétés non fonctionnelles d'un système co-modélisé en Marte et décris par des horloges abstraites. Nous estimons d'abord les valeurs de fréquences des différents processeurs impliqués dans l'exécution. Une fois les fréquences fixées, le temps d'exécution pour chaque processeur est calculée. Ensuite, nous calculons la consommation d'énergie dynamique totale du système. Ayant les fréquences des processeurs, les temps d'exécution et la consommation d'énergie dynamique, nous sommes en mesure d'évaluer la configuration d'une application, d'une architecture et de l'allocation entre une application et une architecture.

7.4.1 Estimation de fréquences minimales de processeurs

La fréquence d'un processeur est le nombre de cycle d'horloge (*cn*), effectués par un processeur en une seconde. En augmentant ou en diminuant la fréquence d'un processeur, le temps d'exécution d'une tâche donnée augmentera ou diminuera respectivement. De notre perspective, la valeur optimale de la fréquence d'un processeur, que nous appelons fréquence minimale, est celle qui assure le traitement de la quantité de travail (*ct*) avant le dépassement des échéances imposées sur la tâche (*dl*) :

$$f_{min} = \frac{ct}{dl} \tag{7.9}$$

Cette hypothèse est vraie pour une exécution mono-processeur. Elle est aussi vraie pour le cas d'une exécution multiprocesseur, où les différents processeurs fonctionnent indépendamment l'un de l'autre. Cependant, quand des dépendances de données existent entre des processeurs, des contraintes de précédences et de périodicités doivent être prises en compte. Ces contraintes reflètent le problème de communication inter-processeurs. Pour cela, les meilleures valeurs de fréquences des processeurs sont celles qui assurent les contraintes temporelles imposées sur le fonctionnement d'un système tout en diminuant autant que possible la fréquence de fonctionnement.

La diminution de la fréquence de processeurs a un impact fort sur la puissance dynamique dissipée du système. Pour cela, il est préférable de diminuer le plus possible la fréquence d'un processeur afin d'avoir une basse consommation d'énergie. Bien évidement, la diminution de la fréquence du processeur est contraignante. Par exemple, la diminution de la vitesse d'exécution d'un processeur pourra engendrer un dépassement de l'échéance imposée sur l'exécution d'une tâche. Cela n'est pas acceptable dans des systèmes temps réel stricts.

7.4.2 Estimation du temps d'exécution

Dans les systèmes embarqués temps réel, le bon fonctionnement du système ne dépend pas uniquement de l'exactitude des valeurs de sortie, mais aussi de la date de délivrance de ces valeurs. Dans la majorité des applications temps réel, une échéance est imposée sur l'exécution d'une tâche ou de toute l'application. Ainsi, il est crucial pour le concepteur système de s'assurer, durant la phase de conception, que les ressources physiques allouées à l'exécution des fonctionnalités sont capables de satisfaire ses contraintes temporelles.

Dans la section précédente, nous avons définis le concept de *fréquence minimale*. Nous entendons par cela la fréquence qui permet, d'une part de prolonger autant que possible la durée d'exécution, et d'autre part, de respecter les contraintes de précédences, de périodicités et les échéances imposées sur un système.

L'estimation du temps d'exécution d'une application peut être calculée en se basant sur le modèles d'horloges abstraites. Nous considérons le temps d'exécution comme un critère supplémentaire pour choisir la meilleure configuration de l'architecture et de l'association. En effet, un ou plusieurs processeurs avec des différentes configurations peuvent satisfaire, d'une part, les propriétés de précédences et de périodicités, et d'autre part, les échéances imposées sur la terminaison du calcul. Pour cela, sélectionner une configuration où les durées d'exécutions tendent vers l'échéance semble être un critère intéressant parmi plusieurs.

Afin d'estimer le temps d'exécution total du système, nous devons prendre en compte la quantité de travail et les cycles *idle* que chaque processeur doit exécuter. Ces informations sont contenues dans les horloges d'exécution, représentées par le nombre

de cycle processeur *cn*. Étant donné un processeur *Proc* ayant une fréquence de calcul f et exécutant *cn* cycles processeur, le temps d'exécution des fonctionnalités $ExecTime$, exprimé en seconde, peut être estimé par la formule suivante :

$$ExecTime = \frac{cn}{f} \tag{7.10}$$

Dans la Figure 7.7 par exemple, $Tclk_1$ nécessite douze cycles de l'horloge associée au processeur `Proc1` ayant une fréquence de 30 MHz. Par conséquent, le temps d'exécution des fonctionnalités associées à l'horloge $Tclk_1$ est de : $12 \times \frac{1}{30} = 0.4\ microsecondes$.

7.4.3 Estimation de la consommation d'énergie

Un des objectifs que nous visons par notre analyse temporelle est de minimiser l'ensemble des configurations possibles de systèmes co-modélisés en Marte. Un des critères essentiels pour juger une configuration est sa consommation d'énergie, l'objectif étant toujours de l'optimiser. Des techniques de diminution de la consommation d'énergie, telles que la DVFS–*Dynamic Voltage Frequency Scaling*, ont été développées afin de changer dynamiquement la quantité de travail d'un processeur durant son activité. Cela se fait en général en réglant les valeurs de fréquences/tensions des processeurs selon la grandeur de la quantité de travail à accomplir. Cette modification doit prendre en compte bien évidement les contraintes d'échéances qui peuvent être imposées sur des tâches fonctionnelles [82].

Nous nous inspirons de la technique DVFS afin de diminuer autant que possible la consommation d'énergie du système. Compte tenu de la spécification de fonctionnalités et de leurs associations sur des architectures multiprocesseur, nous procédons par fixation des valeurs des fréquences/tensions de processeurs. Cela est possible en analysant statiquement la quantité de travail allouée à chaque processeur. En choisissant des fréquences minimales, nous augmentons au maximum le taux d'utilisation des processeurs (l'idéal étant d'arriver à une utilisation de 100% pour chaque processeur).

La quantité d'énergie dissipée par un processeur lors de son utilisation est composée d'une partie statique et d'une autre dynamique. L'énergie statique dissipée est principalement due aux fuites de courants. Celle-ci peut être évitée quand un processeur est éteint. La consommation d'énergie dynamique est directement liée aux valeurs de la fréquence/tension du processeur. Dans le cadre de cette thèse, nous ne considérons que l'énergie dynamique. Dans le cas d'un modèle d'horloges abstraites, nous définissons l'énergie totale dissipée d'un système selon la formule suivante :

$$E_{total} = \sum_{x=1}^{x=n} E_x = \sum_{x=1}^{x=n} (P_x \times t_x) \tag{7.11}$$

où n représente le nombre total d'horloges d'exécutions associées aux processeurs impliqués dans l'exécution, P_x représente la puissance dissipée et t_c son temps d'exécution. La puissance dynamique dissipée est cependant dépendante de trois variables, et est définie par la formule suivante :

$$P = C \times f \times V^2 \tag{7.12}$$

où C représente la capacité (une valeur fournit par le constructeur de la puce), f est la fréquence du processeur et V est la tension. Dû à une relation linéaire entre la valeur de la fréquence et de la tension d'un processeur, réduisant ou augmentant la valeur de la

fréquence affecte linéairement la valeur de la tension. Avec ces valeurs, nous sommes en mesure de déduire la quantité d'énergie dissipée du système.

La question **Q3**, posée à la fin du chapitre 1 de l'introduction, exprime le besoin d'intégrer une méthodologie d'analyse, de haut niveau, de propriétés fonctionnelles et non fonctionnelles d'applications hautes performances implantées sur MPSoC. En effet, en augmentant le niveau d'abstraction, nous obtenons une rapidité dans l'analyse. Cela permet d'explorer un grand ensemble de configurations possibles de systèmes. La motivation de cette analyse est donc de réduire cet ensemble en un nombre limité de configurations, jugées idéales selon les critères du concepteur.

Nous avons proposé dans ce chapitre une méthodologie d'analyse et de vérification de propriétés fonctionnelles et non fonctionnelles, par le biais d'horloges abstraites. La méthodologie proposée repose sur le modèle Y en faisant une séparation claire entre les fonctionnalités et les architectures considérées. Nous avons proposé trois types d'horloges abstraites : fonctionnelle, physique et d'exécution. Les horloges fonctionnelles abstraient des applications. Les horloges physiques tracent les vitesses de processeurs. Finalement, les horloges d'exécutions simulent l'activité des différents processeurs durant l'exécution de fonctionnalités. Nous avons réussi à valider des contraintes fonctionnelles, telles que l'ordre d'exécution de tâches. Nous avons aussi réussi à analyser des propriétés non fonctionnelles telles que l'estimation des temps d'exécution et de la consommation totale d'énergie. La rapidité des analyses et de vérifications fait de notre proposition un travail innovateur.

7.5 Conclusion

Nous avons présenté dans ce chapitre une technique d'analyse de propriétés fonctionnelles et non fonctionnelles de MPSoC. Ces MPSoC sont caractérisés par la manipulation d'applications hautes performances, contenants de grandes quantités de données, ainsi que des architectures multiprocesseur massivement parallèles.

Les systèmes, co-modélisés en Marte, sont décrits par des horloges abstraites. Ces dernières permettent une abstraction de fonctionnalités, d'architectures et de l'association entre des applications et des architectures. L'analyse d'horloges abstraites nous permet d'analyser et de vérifier, rapidement, plusieurs configurations d'architectures et d'associations pour un système. La contribution majeure de ce travail est la facilité de manipulation des horloges abstraites ainsi que la rapidité de l'analyse et de la vérification (le temps étant négligeable par rapport à d'autres approches d'analyse et de simulations à des niveaux plus bas tels que TLM ou RTL). C'est un travail novateur car les travaux récents, à partir du niveau de détails choisis, visent uniquement la conception des systèmes matériels/logiciels sans prendre en compte une phase d'analyse, de vérification et d'exploration de l'espace de conception. Nous avons pu prouvé qu'une phase d'exploration de l'espace de conception dans ses premiers stades dans le flot de conception s'avère très utile pour réduire l'espace de solutions possibles.

Pour plus de souplesse de conception et d'utilisation, la tendance actuelle est d'automatiser les étapes d'analyses et de vérification sous forme d'un outil. Afin de tester un grand espace de solutions possibles de configurations, nous envisageons un branchement de cet outil dans un environnement d'exploration de systèmes. Cela offrira un grand choix de test sur différentes configurations amenant ainsi à un choix d'architecture et d'allocation optimale vis-à-vis la validation des contraintes temporelles et d'une consommation d'énergie optimale.

Part III

Intégration dans Gaspard2 et validation sur une étude de cas

Chapter 8

Un noyau de manipulation d'horloges pour une intégration dans Gaspard2

8.1 Introduction

Dans les chapitres précédents nous avons présenté les différentes contributions de cette thèse. Au fur et à mesure, des exemples ciblés sur quelques notions particulières ont illustré ou validé en partie ces contributions.

Dans ce chapitre, nous proposons un noyau de manipulation d'horloges abstraites afin d'être intégré ultérieurement dans Gaspard2. Ce noyau repose sur les techniques d'analyse, proposées dans cette thèse. La Figure 8.1 montre le processus automatique d'analyses de systèmes. D'abord une co-modélisation d'un système est faite par le biais du profil Marte. Cette modélisation comporte une description d'une application, d'une architecture et d'une association entre les deux. Une fois le modèle Marte obtenu, il est transformé automatiquement en un modèle intermédiaire contant des horloges fonctionnelles et physiques. Dans une étape suivante, l'algorithme d'ordonnancement de tâches, basé sur des horloges abstraites, permet de projeter des horloges fonctionnelles sur des horloges physiques. Le résultat de cette projection est un ensemble d'horloges d'exécution. L'ensemble de ces horloges représente des entrées pour un outil, basé sur des fonctions OCaml. Cet outil permet l'analyse et la vérification automatique de couples d'horloges abstraites. Les résultats de l'évaluation du modèle Marte sont ensuite retournés au concepteur afin de raffiner les choix de configurations.

Dans la section 8.2, nous proposons un outillage qui manipule des horloges abstraites. Il permet d'analyser des couples d'horloges abstraites fonctionnelles ou d'exécution.

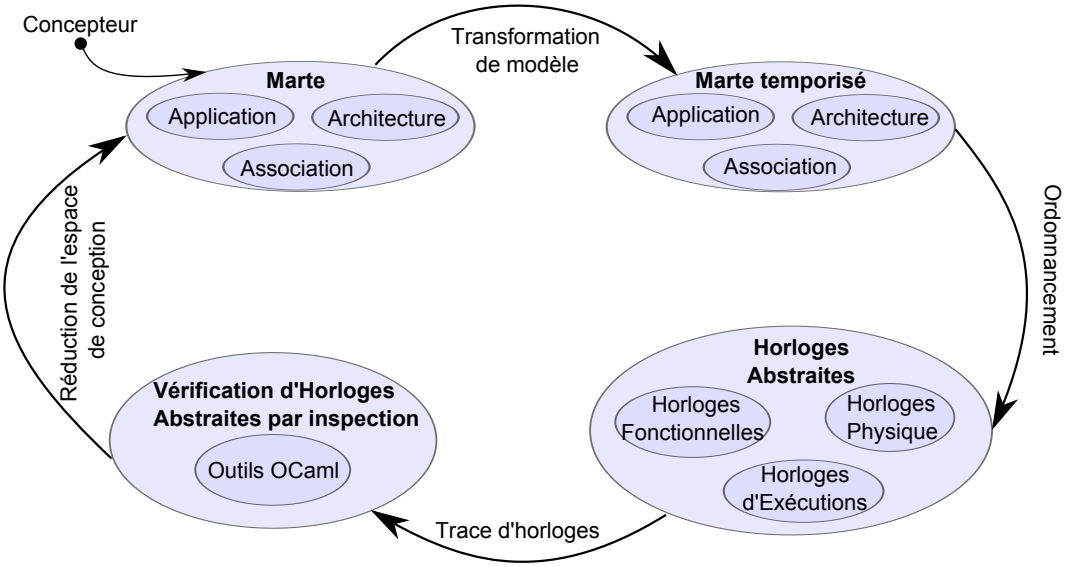


Figure 8.1: Description du noyau de manipulation d'horloges abstraites.

Dans la section 8.3, nous proposons une extension du métamodèle Marte afin de pouvoir associer des horloges physiques à des processeurs et des horloges fonctionnelles à des tâches. La section 8.4 propose une technique de transformation de modèles, d'un modèle Marte non temporisé, vers un modèle Marte contenant des notions d'horloges abstraites. Enfin, nous concluons dans la section 8.5.

8.2 Un prototype de bibliothèque de manipulation d'horloges

Nous avons proposé au cours de cette thèse une co-modélisation, en Marte, d'applications hautes performances implantées sur de MPSoC. Nous avons aussi proposé une méthodologie d'analyse et de vérification de systèmes, à base d'horloges abstraites, afin d'évaluer différentes configurations possibles d'applications, d'architectures et d'associations. Cependant, nous n'avons pas trouvé un outil qui permet la spécification et la manipulation efficace de ce type d'horloges. C'est pour cela que nous avons implémenté un prototype en se servant du langage fonctionnel OCaml [88].

Le langage OCaml

Nous avons choisi le langage OCaml pour implémenter la méthodologie d'analyse et de vérification d'horloges abstraites pour plusieurs raisons :

- OCaml est un langage de haut niveau qui permet de concevoir des prototypes rapidement et d'appliquer des modifications facilement ;
- le typage fort en OCaml assure la détection de beaucoup d'erreurs ;
- les exécutables générés sont (en général) très efficaces et facilement utilisables.

OCaml est, par ailleurs, disponible pour un grand nombre de plateformes (architectures et systèmes d'exploitations). Les sources binaires du compilateur ainsi qu'une documentation très conviviale sont disponibles sur Internet [88].

Nous proposons donc un prototype d'analyse et de vérification de systèmes, basé sur des fonctions CAML. Dans ce formalisme, nous manipulons des couples d'horloges fonctionnelles. Ces dernières indiquent des ordres d'exécutions de tâches fonctionnelles. Nous manipulons aussi des horloges d'exécution, afin de vérifier des propriétés fonctionnelles et non fonctionnelles. Ces deux types d'horloges portent des valeurs bien définies :

- les instants logiques d'une horloge fonctionnelle portent des valeurs de type binaire : une horloge fonctionnelle est représentée par une suite de valeurs "0" et/ou "1" ;
- les instants logiques d'une horloge d'exécution portent des valeurs de type ternaire : une horloge d'exécution est représentée par une suite de valeurs "0", "1" et/ou "-1".

Ces deux types d'horloges abstraites représentent une abstraction de systèmes, comodélisés en Marte. Elles nous seront utiles pour détecter des violations potentielles de l'ordre d'exécution de tâches. Notamment nous vérifions le respect de contraintes de périodicités et/ou de précédences. Nous utilisons aussi des horloges d'exécution afin de vérifier le respect des échéances, imposées sur la terminaison des exécutions de tâches.

8.2.1 Format des entrées

L'outil prend en entrée des couples d'horloges fonctionnelles ou d'exécution dont les instants sont respectivement de types binaires et ternaires. Dans les deux cas, une horloge a le format d'une liste de valeurs $i \in \mathbb{Z}$.

$$clk_1 = [1;1;1;0;1;0;1;0;0;1;0;1;0;0;0;1]$$
$$clk_2 = [0;1;0;0;0;1;0;1;1;1;1;0;0;1;1]$$

Figure 8.2: Un exemple de deux horloges fonctionnelles clk_1 et clk_2, représentées sous forme de listes, et associées respectivement aux tâches C_1 et C_2.

Quand les horloges manipulées sont de types fonctionnelles, l'outil prend en entrée des couples d'horloges (clk_i,clk_j), où clk_i et clk_j sont associées respectivement à des composants élémentaires C_i et C_j. Nous considérons toujours C_i comme étant un composant producteur de données et C_j un composant consommateur de données, les deux étant en relation direct. Dans le modèle Marte correspondant, C_i est relié avec C_j par un connecteur. Les deux horloges fonctionnelles clk_1 et clk_2, illustrées dans la Figure 8.2, sont un exemple de format d'entrée pour une analyse de couple d'horloges abstraites fonctionnelles. Dans cet exemple, les composants C_1 et C_2 sont activés huit fois.

$$TClk_1 = [-1;1;-1;1;0;1;-1;1;0;1;-1;1;0]$$
$$TClk_2 = [-1;-1;-1;1;1;-1;0;1;0;1;-1;-1;1]$$

Figure 8.3: Un exemple de deux horloges d'exécution $TClk_1$ et $TClk_2$, représentées sous forme de liste, et associées respectivement aux processeurs $Proc_i$ et $Proc_j$.

De la même manière, quand les horloges manipulées simulent l'exécution de processeurs (ou horloges d'exécution), l'outil prend en entrée des couples d'horloges ($TClk_i$,$TClk_j$). La Figure 8.3 montre un exemple de deux horloges d'exécution $TClk_1$

et $TClk_2$, considérées comme entrées pour l'outil, afin d'analyser l'exécution des deux processeurs associés $Proc_1$ et $Proc_2$.

Le processeur $Proc_1$ effectue du traitement de données et sauvegarde le résultat dans une mémoire partagée. Le processeur $Proc_2$ effectue aussi du traitement de données. Cependant, ce traitement est dépendant de la vitesse de traitement du processeur $Proc_1$.

8.2.2 Définition des algorithmes

Nous avons défini une bibliothèque de fonctions OCaml permettant d'analyser des couples d'horloges. Ces fonctions visent principalement les objectifs suivants :

- dimensionnement de mémoires nécessaires pour assurer des communications, sans perte ou écrasement de données ;
- analyse des communications entre deux processeurs au niveau exécution, selon les contraintes de précédences, de périodicités et des temps d'échéances imposées.

Cette bibliothèque de fonctions peut être téléchargée à partir de la page web :

"http://www.lifl.fr/~abdallah/Analyse_Synchronisation.tgz"

Les horloges sont analysées dans un style fonctionnel. Chaque horloge à la forme d'une liste. Cela permet d'obtenir, d'une part, des descriptions concises et, d'autre part, une bonne lisibilité des instants d'activation dans une horloge.

Localisation des activations sur une horloge

Nous nous intéressons dans cette section à l'analyse temporelle d'événements du type "*une activation dans une horloge clk_i arrive avant une autre activation dans une horloge clk_j*". Dans ce que nous proposons, seules les occurrences de la valeur "1" nous intéressent :

- l'occurrence d'une valeur 1 dans une horloge abstraite fonctionnelle indique l'activation d'un composant ;
- l'occurrence d'une valeur 1 dans une horloge abstraite d'exécution indique le début de traitement pour un processeur.

Les positions des valeurs "1" dans une horloge sont donc de grandes importances. Pour cela, nous définissons une fonction *activation_occurrence*() qui parcoure les instants d'une horloge, en mémorisant les positions des instants portant la valeur "1". Cette fonction prend en entrée une horloge clk_i, sous forme d'une liste, et rend en sortie des valeurs $x_j \in \mathbb{N}$, où $j \in [1, taille(clk_i)]$, aussi sous forme d'une liste.

En appliquant cette fonction sur les horloges clk_1 et clk_2 de l'exemple précédent, nous obtenons les deux nouvelles listes l_1 et l_2, illustrées par la Figure 8.4.

$$l_1 = activation\_occurrence(clk_1) = [1; 2; 3; 5; 7; 10; 12; 14]$$
$$l_2 = activation\_occurrence(clk_2) = [2; 6; 8; 9; 10; 11; 14; 15]$$

Figure 8.4: Application de la fonction *activation_occurrence*() sur les horloges clk_1 et clk_2.

En appliquant cette fonction sur n listes, représentants des horloges abstraites, nous obtenons en sortie l_i listes, où $i \in [1, n]$. Chacune des liste l_i contient les positions des instants de l'horloge clk_i portant la valeur "1". Cela nous permet de vérifier, dans une étape suivante, des contraintes de périodicités, de précédences et de temps d'échéances.

Contraintes de dépendances de tâches

Des contraintes de précédences et de périodicités peuvent être imposées sur des horloges abstraites fonctionnelles ou d'exécution. Nous souhaitons prendre en considération de telles contraintes, imposées sur des couples d'horloges abstraites. Par exemple, chaque occurrence d'une valeur "1" dans une horloge clk_j doit être précédée par trois occurrences de valeurs "1" dans une horloge clk_i. Nous appelons *Contrainte*1 cette contrainte.

Nous définissons la fonction $slice\_activation\_occurrence(l_i, \alpha_i)$, où l_i est une liste contenant des valeurs $x_m \in \mathbb{N}$ et $\alpha_i \in \mathbb{N}$ est une variable représentant la contrainte imposée sur l'horloge clk_i par rapport à une autre horloge clk_j. Le résultat de l'application de cette fonction sur une horloge clk_i est une nouvelle liste $ll_i = [[sl_i^1], [sl_i^2], ..., [sl_i^x]]$[1], où $x = Int(\frac{taille(clk_i)}{\alpha_i})$. Cette fonction divise la liste l_i en des sous listes sl_i^j, toutes de taille α_i. Ensuite, cette fonction transforme chaque sous liste sl_i^j en une sous liste $sl_i'^j$, de la manière suivante :

$$sl_i'^j = [borne\_suprieur(sl_i^j); borne\_infrieur(sl_i^j)] \tag{8.1}$$

Le résultat de cette transformation est une liste ll_i' de sous listes.

Exemple : nous considérons les deux listes l_1 et l_2, représentants respectivement les occurrences d'activations des horloges clk_1 et clk_2. En appliquant cette fonction sur les listes l_1 et l_2, nous obtenons les sous listes suivantes :

- $ll_1 = [[1; 2; 3]; [5; 7; 10]; [12; 14]]$;
- $ll_1' = slice\_activation\_occurrence(l_1, 3) = [[3; 1]; [10; 5]; [14; 12]]$;
- $ll_2 = [[2]; [6]; [8]; [9]; [10]; [11]; [14]; [15]]$;
- $ll_2' = slice\_activation\_occurrence(l_2, 1) = [[2]; [6]; [8]; [9]; [10]; [11]; [14]; [15]]$.

Nous avons ainsi réorganisé les listes l_1 et l_2 selon la contrainte *Contrainte*1. Nous avons regroupé les $n^{\text{ième}}$ trois occurrences de 1 successives dans clk_1 dans une sous liste $sl_1''^n$ et la $n^{\text{ième}}$ occurrence de 1 dans clk_j dans une sous liste $sl_2''^n$.

Afin dé vérifier les contraintes de dépendances de tâches entre deux horloges clk_i et clk_j, il suffit de comparer les sous listes de chacune de deux horloges. Si la plus petite valeur de la $n^{\text{ième}}$ sous liste $sl_j''^n$ est plus grande ou égale à la plus grande valeur de la $n^{\text{ième}}$ sous liste $sl_i''^n$, cela signifie que les contraintes sont vérifiées. En d'autres termes, les sous listes $sl_i''^n$ et $sl_j''^n$ doivent vérifier la contrainte suivante :

$$Min(sl_j''^n) - Max(sl_i''^n) \geq 0 \tag{8.2}$$

Dans le cas contraire, les contraintes de dépendances de tâches sont violées.

Par exemple, la première sous liste $sl_1'^1$ de la liste ll_1' est [3;1] et la première sous liste $sl_2'^1$ de la liste ll_2' est [2]. En appliquant l'équation 8.2, nous remarquons que $Min(sl_2'^1) - Max(sl_1'^1) = -1$. Cela signifie que la tâche T_j est activée avant que trois activations de la tâche T_i n'aient eu lieu. Donc la contrainte *Contrainte*1 n'est pas respectée.

[1] *ll* est une abréviation de *liste de liste* et *sl* est une abréviation de *sous liste*

Estimation de tailles minimales de mémoires tampons

Nous définissons une fonction de synchronisation qui peut être appliquée sur des couples d'horloges abstraites. Cette fonction est une adaptation de la technique classique de synchronisation d'horloges, définie par Cohen et al. [29]. Elle permet d'analyser la connexion entre deux composants communicants et de déterminer des mémoires tampons là où cela s'avère nécessaire.

Definition 18 (Taille minimale de Mémoire tampon) *Une taille minimale d'une mémoire tampon, insérée entre deux tâches communicantes, est égale à la plus grande quantité de données générées et non pas encore consommées, depuis le début d'un traitement jusqu'à ca fin.*

Étant donné des couples d'horloges, vérifiant les contraintes de dépendances de tâches, il est possible de déterminer la taille minimale de mémoire tampon nécessaire pour mémoriser les données générées et non pas encore consommées. En effet, les couples d'horloges (clk_i,clk_j), considérés comme entrées pour le prototype d'analyse, représentent respectivement des tâches (ou processeurs) productrices et consommatrices de données. Ainsi, dans le formalisme d'analyse d'horloges abstraites que nous avons proposé à l'aide de fonctions OCaml, il suffit d'appliquer la formule 8.2 sur l'ensemble des sous listes de taille t et ensuite de choisir la valeur maximale des résultats :

$$Buffer_{Min}(sl'_i, sl'_j) = Max(Min(sl'^n_j) - Max(sl'^n_i)) \tag{8.3}$$

où $n \in [1, t]$.

Prenons maintenant comme example les deux horloges $TClk_1 = [0; -1; 1; 1; 0; 1; 1; -1; 1; 1; 0]$ et $TClk_2 = [0; -1; -1; -1; 0; 1; -1; -1; -1; 0; 1]$. L'horloge $TClk_1$ (resp. $TClk_2$) est associée à un processeur $Proc_1$ (resp. $Proc_2$) producteur (resp. consommatrice) de données. Ces deux horloges vérifient l'ordre d'exécution de tâches imposées par la contrainte *Contrainte*1. En effet, en localisant les activations de tâches sur les deux horloges, nous obtenons les listes $l_1 = activation\_occurrence(TClk_1) = [3; 4; 6; 7; 9; 10]$ et $l_2 = activation\_occurrence(TClk_2) = [6; 11]$. Nous appliquons ensuite le découpage des listes l_1 et l_2 selon la contrainte *Contrainte*1. Nous obtenons $sl'_1 = slice\_activation\_occurrence(l_1, 3) = [[6; 3]; [10; 7]]$ et $sl'_2 = slice\_activation\_occurrence(l_2, 1) = [[6]; [11]]$. En appliquant la formule 8.2 sur les sous listes sl'_1 et sl'_2, nous remarquons que les deux horloges $TClk_1$ et $TClk_2$ respectent la contrainte *Contrainte*1 :

$$Min(sl'^1_j) - Max(sl'^1_i) = 0$$
$$Min(sl'^2_j) - Max(sl'^2_i) = 1$$

Nous avons donc besoin d'une mémoire tampon de taille $Buffer\_Min = Max(0, 1) = 1$.

Afin de connecter l'ensemble des fonctions l'une avec l'autre, nous définissons la fonction $validate\_clkProducer\_clkConsumer(clk_{i,c}\, lk_j, \alpha_i, \alpha_j, dl)$, où clk_i et clk_j représentent respectivement deux horloges abstraites, α_i et α_j les contraintes de dépendances de tâches respectivement sur clk_i et clk_j et dl le temps d'échéance imposé sur la terminaison de l'exécution. L'application de cette fonction revient à appliquer les fonctions imbriquées suivantes :

$$Buffer(slice\_activation\_occurrence(activation\_occurrence(clk_i), \alpha_i)\ ,$$
$$slice\_activation\_occurrence(activation\_occurrence(clk_j), \alpha_j))$$

La Figure 8.5 montre une capture d'écran de l'utilisation du prototype développé en OCaml. Elle consiste en une analyse de synchronisabilité entre deux horloges fonctionnelles clk_1 et clk_2. Nous insérons manuellement, dans une première étape, les valeurs binaires de l'horloge clk_1 (première occurrence du symbole # sur la Figure 8.5). De la même

manière, nous insérons les valeurs binaires de l'horloge clk_2 (deuxième occurrence du symbole #). Ensuite, nous appliquons la fonction *validate_clkProducer_clkConsumer*(clk_1, $_clk_2$, 1, 2, 10) qui représente une analyse de synchronisation des deux horloges clk_1 et clk_2 (troisième occurrence du symbole #). Deux contraintes sont considérées dans l'analyse :

1. une contrainte de rythme d'horloge impose que l'horloge clk_2 soit deux fois plus rapide que clk_1 ;

2. un deadline de dix instants logiques est imposé sur la terminaison du calcul.

Les résultats obtenus, à partir de l'analyse des deux horloges avec le prototype, indiquent que la communication entre les deux horloges clk_1 et clk_2 est synchronisable par une insertion d'une mémoire tampon de taille 2. Cependant, le temps d'échéance (de valeur 10) imposé n'est pas respecté. Il a été dépassé de 63 instants logiques sur clk_1 et de 64 instants logiques sur clk_2.

La même analyse est appliquée de nouveau sur les deux horloges clk_1 et clk_2 (quatrième occurrence du symbole #) mais en fixant cette fois-ci le temps d'échéance à 1000.

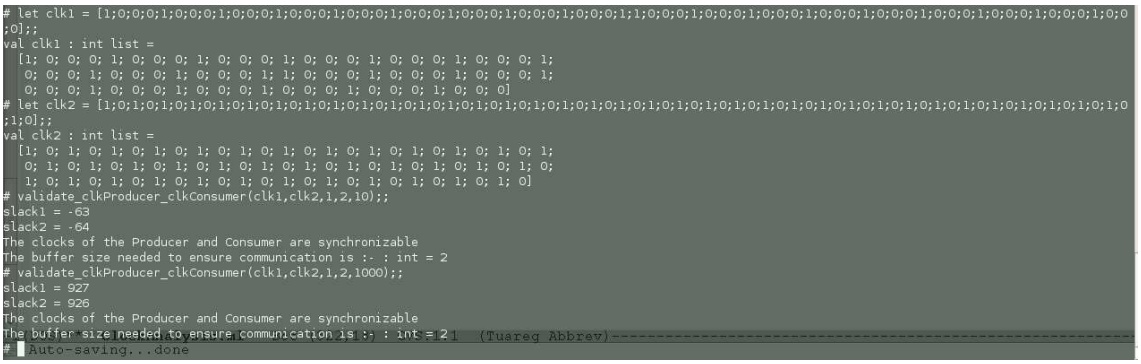


Figure 8.5: Une capture d'écran de l'utilisation du prototype pour l'analyse de synchronisabilité de deux horloges clk_1 et clk_2.

Un des point essentiel de notre approche est le fait que les horloges sont nécessairement décrites explicitement de manière énumératives. Nous commençons par raffiner la grande taille des horloges abstraites en analysant juste les instants portant la valeurs "1" sur chaque horloge. Cela permet de réduire la taille d'horloges manipulées.

8.3 Métamodèle intermédiaire pour la génération des inputs

Nous avons proposé dans cette thèse une analyse de systèmes, co-modélisés en Marte, par le biais d'horloges abstraites. Ces horloges sont classifiées comme suit :

- fonctionnelle : les horloges fonctionnelles abstraient une application Marte, en spécifiant les activations des différents composants élémentaires ainsi que les dépendances de tâches sous-jacentes (représentant un ordre d'exécution) ;

- physique : dans une modélisation Marte d'une architecture MPSoC, une horloge physique est associée à chaque processeur. Cette horloge à une période égale à l'inverse de la fréquence du processeur concerné ;

- exécution : quand une application, une architecture et leur association sont modélisées en ' Marte, les horloges d'exécutions ont pour but de simuler l'activité des différents processeurs impliquées dans l'exécution.

Les horloges d'exécutions sont obtenues en appliquant des projections d'horloges fonctionnelles sur des horloges physiques. Il est donc nécessaire d'avoir l'ensemble des horloges fonctionnelles et physiques afin de pouvoir générer les horloges d'exécutions.

Pour faciliter la génération automatique d'horloges fonctionnelles et physiques, nous étendons le métamodèle Marte afin de pouvoir ajouter les informations suivantes :

- dans une application modélisée en Marte, nous associons à chaque instance de composant élémentaire une horloge abstraite fonctionnelle. Cette horloge doit contenir les informations suivantes :
  - le nombre de répétition total de l'instance de composant élémentaire qui lui est associée ;
  - un lien vers l'horloge fonctionnelle, associée au composant générateur de données ;
  - un lien vers l'horloge fonctionnelle, associée au composant consommateur de données ;
  - l'horloge physique qui lui est associée.
- dans une architecture modélisée en Marte, nous associons à chaque instance d'un processeur une horloge physique. Cette horloge doit aussi contenir les informations suivantes :
  - la fréquence du processeur associée, afin de tracer les périodes de l'horloge ;
  - l'ensemble des horloges fonctionnelles exécutées sur le processeur associé ;

Dans la Figure 8.6, nous proposons un métamodèle intermédiaire qui représente une extension du métamodèle Marte. Ce métamodèle contient des notions d'horloges abstraites.

La classe `AssemblyPart` est un concept de Marte. Un `AssemblyPart` permet de définir des instances de composants physiques (processeur, mémoire, bus, etc.) et des instances de composants fonctionnels. La classe `PhysicalClock` représente une horloge abstraite physique alors que `LogicalClock` représente une horloge abstraite fonctionnelle. Ces deux horloges sont contenues dans la classe `AssemblyPart`.

Ainsi, chaque instance de processeur, définie comme étant un `AssemblyPart`, contiendra une horloge physique `PhysicalClock`. De la même manière, chaque instance de composant fonctionnelle, définie comme étant un `AssemblyPart`, contiendra une horloge logique `LogicalClock`. Les deux horloges `PhysicalClock` et `LogicalClock` ont un attribut `value` du type entier. Pour une horloge physique, l'attribut `value` indique la fréquence du processeur associé. Dans la cas d'une horloge logique, l'attribut `value` indique le nombre de répétition de l'instance de composant associée à cet horloge.

La classe `LogicalClock` contient un connecteur cyclique dont les extrémités sont `previous` et `next`. Ainsi, dans un modèle Marte, une horloge logique du type `LogicalClock` aura deux propriétés `previous` et `next` indiquant respectivement les horloges logiques des composants générateurs de données et les horloges logiques des composants consommateurs de données.

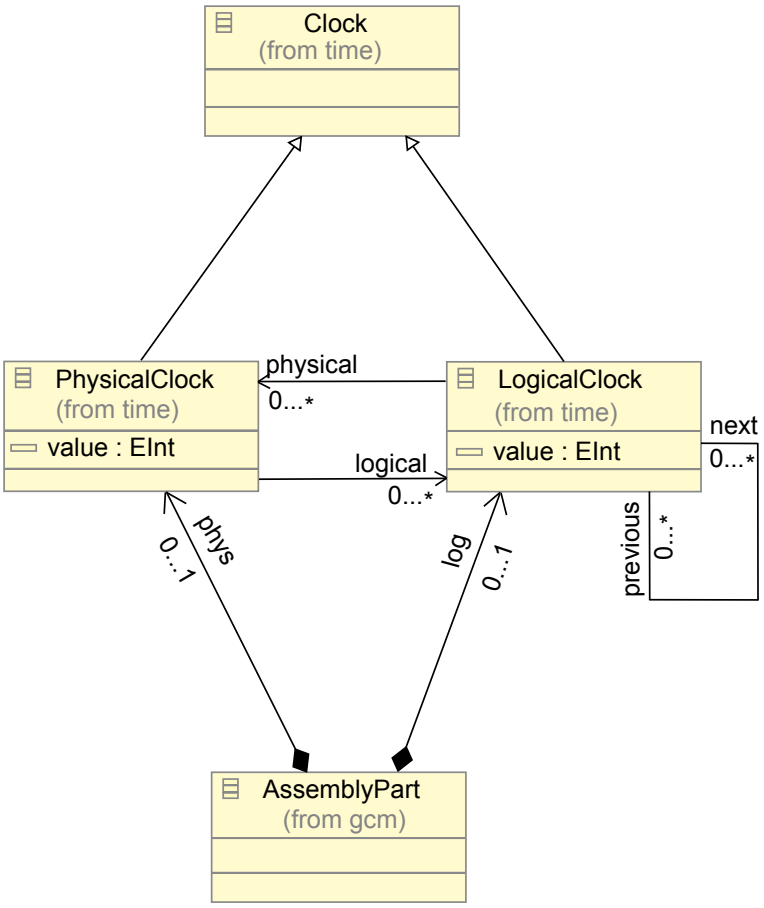


Figure 8.6: Extension du métamodèle Marte avec des notions d'horloges abstraites.

8.4 Transformation d'un modèle Gaspard2 vers un modèle intermédiaire

Nous définissons une transformation de modèle, écrite en qvto, afin de générer automatiquement, à partir d'un modèle Marte non temporisé, un modèle Marte contenant des horloges abstraites fonctionnelles et physiques. Cette transformation est de type 1..1 :

- elle prend en entrée un modèle Marte, conforme au métamodèle Marte ;
- elle génère en sortie un modèle Marte temporisé, conforme au métamodèle intermédiaire.

La Figure 8.7 montre un exemple d'un modèle Marte décrivant un *downscaler*. Ce modèle est conforme au métamodèle Marte. La classe `MainApplication`, du type `Structured Component` (un concept de Marte), contient trois classes fonctionnelles de types `Assembly Part` :

- `Prod` : elle représente une instance d'un composant élémentaire `Producer` qui produit des données ;
- `Downscale` : elle représente une instance du composant composé `Downscaler` qui applique l'algorithme du downscaler ;
- `Cons` : elle représente une instance du composant élémentaire `Consumer` qui lit des données ;

Figure 8.7: Exemple de fonctionnalités d'un modèle *downscaler*, conforme au métamodèle Marte.

En appliquant une transformation de modèle, sur le modèle *downscaler*, nous obtenons en sortie un modèle conforme au métamodèle intermédiaire. Ce modèle de sortie, illustré par la Figure 8.8, contient des horloges abstraites fonctionnelles. Nous remarquons que deux horloges fonctionnelles ont été créées pour les classes `Prod` et `Cons` de type `AssemblyPart` alors que la classe `Downscale` est vide. En effet, dans la méthodologie d'analyse proposée, nous ne considérons que les instances de composants élémentaires.

L'attribut `Next` pointe vers l'horloge fonctionnelle `Logical_Clk_VFilter`. L'attribut `Previous` est vide. Cela signifie que c'est le premier composant élémentaire dans le graphe de composant élémentaire et donc ne possède pas un prédécesseur. L'attribut `Physical` pointe vers l'horloge physique `Physical_Clk_Proc1`.

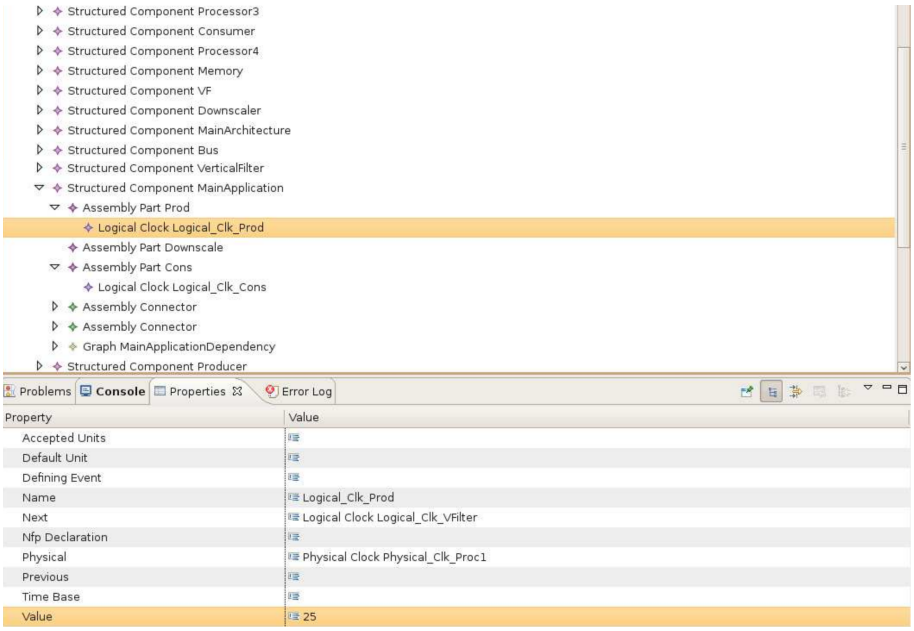


Figure 8.8: Modèle du *downscaler*, conforme au métamodèle intermédiaire, généré automatiquement par une transformation de modèle et contenant des horloges fonctionnelles.

De la même manière, la Figure 8.9 illustre le modèle du *downscaler* en focalisant sur l'architecture multiprocesseur considérée. La classe `MainArchitecture`, de type `StructuredComponent`, contient six classes de type `AssemblyPart`. Parmi ces six classes, quatre d'entre elles (`Proc1`, `Proc2`, `Proc3` et `Proc4`), stéréotypées `hwprocessor`, représentent des processeurs.

En appliquant une transformation de modèle, du modèle *downscaler* vers un modèle intermédiaire, nous obtenons des modifications au niveau de l'architecture, illustrées par la Figure 8.10. Les quatre processeurs `Proc1`, `Proc2`, `Proc3` et `Proc4` contiennent respectivement les horloges physiques `Clock_Physical_Proc1`, `Clock_Physical_Proc2`, `Clock_Physical_Proc3` et `Clock_Physical_Proc4`. L'attribut `Logical` de l'horloge physique `Logical_Clk_HFilter`.

La question **Q4**, posée à la fin du chapitre 1 de l'introduction, exprime le besoin d'un outillage permettant d'analyser automatiquement des horloges abstraites. L'objectif est de faciliter et d'accélérer l'analyse d'un ensemble de configurations possibles d'un système.

Dans ce chapitre, nous avons présenté des techniques IDM permettant de faciliter l'analyse temporelle de MPSoC. Les techniques proposées représentent les premières étapes vers l'automatisation complète de la méthodologie d'analyse proposée. Nous répondons ainsi, en partie, à la question **Q4** évoquée dans le chapitre 1.

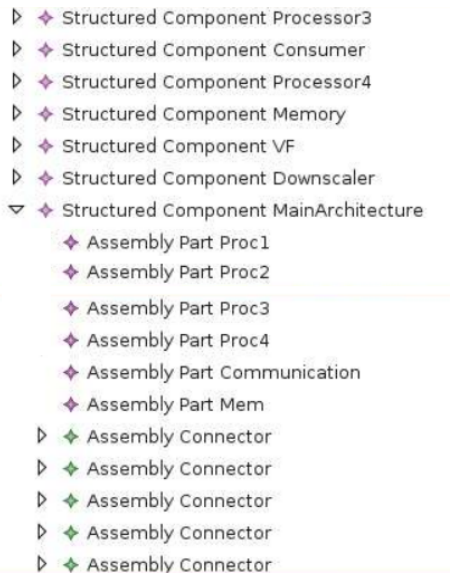


Figure 8.9: Exemple d'architecture d'un modèle *downscaler*, conforme au métamodèle Marte.

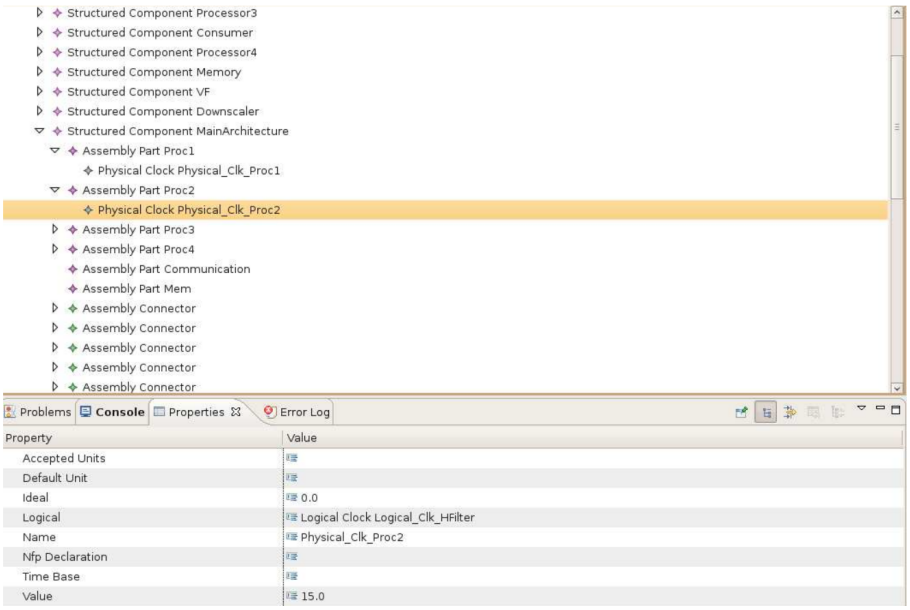


Figure 8.10: Modèle du *downscaler*, conforme au métamodèle intermédiaire, généré automatiquement par une transformation de modèle et contenant des horloges physiques.

8.5 Conclusion

Afin d'expérimenter la mise en pratique de l'analyse temporelle proposée, nous avons conçu un outil composé de plusieurs fonctions écrites en OCaml. Cet outil est capable d'analyser des horloges fonctionnelles et d'exécution. Il offre une étude de synchronisabilité entre des couples d'horloges et permet de vérifier les ordres d'exécution de tâches. Il permet aussi d'estimer des tailles minimales de mémoires tampons nécessaires pour sauvegarder des données générées et non pas encore consommées.

Cependant, l'outil proposé est capable de produire un résultat final d'analyse entre des couples d'horloges explicitement dans le cas des horloges ayant une taille raisonnable (moins de dix milles instants). Si des couples d'horloges dépassent cette taille, l'analyse proposée n'est plus efficace aussi bien au niveau du temps de calcul qu'au niveau occupation mémoire. Donc des optimisations doivent être mise en place afin d'augmenter les performances des algorithmes définis.

Afin d'automatiser la méthodologie proposée, nous avons défini un métamodèle intermédiaire. Ce dernier étend le métamodèle Marte par des concepts d'horloges abstraites fonctionnelles et physiques. Les horloges fonctionnelles sont associées automatiquement à des composants élémentaires décrivant de fonctionnalités alors que les horloges physiques sont associées à des processeurs. Le passage d'un modèle Marte non temporisé vers un modèle intermédiaire temporisé est fait par le biais d'une transformation de modèle écrite en qvto.

L'automatisation complète de l'approche nécessite encore du travail. Elle est parmi les différentes perspectives de cette thèse. En effet, l'étape de projection d'horloges fonctionnelles sur des horloges physiques est encore manuelle. Cette étape, une fois automatisé, permettra une analyse de propriétés fonctionnelles et non fonctionnelles d'un système en quelques secondes de temps du processeur. L'outil, une fois automatisé complètement, doit être branché dans un environnement d'exploration de l'espace de conception afin de trouver la ou les configurations optimales de systèmes. On pourra par exemple envisager un environnement d'exploration à base d'algorithmes génétiques.

Chapter 9

Étude de cas : encodeur JPEG sur une architecture PRAM

9.1 Présentation générale

Nous allons illustré et validé dans ce chapitre les contributions proposées dans cette thèse. Nous proposons une étude de cas pour valider, de manière expérimentale, les différentes contributions dans le processus complet, de la modélisation de MPSoC jusqu'à l'analyse temporelle et la vérification du domaine de conception. C'est également l'occasion pour le lecteur de se faire une idée plus précise sur l'organisation globale de la co-modélisation de MPSoC à l'aide de l'environnement Gaspard2.

Nous allons dans un premier temps présenter un modèle de MPSoC consistant d'un encodeur vidéo JPEG placé sur une architecture multiprocesseur de types PRAM. Nous illustrons ensuite l'utilisation de la méthodologie d'analyse d'horloges abstraites, présentée dans cette thèse, afin de montrer son utilité. Nous vérifions des contraintes fonctionnelles telles que l'ordre d'exécution de tâches (contraintes de périodicités et de précédences) et le respect des temps d'échéances imposés sur la terminaison de travail. Nous analysons aussi des propriétés non fonctionnelles telles que l'estimation de temps d'exécution et de la consommation d'énergie du système. Nous comparons ensuite les résultats obtenus avec des simulations, faites au niveau TLM en SystemC, de l'application JPEG sur une architecture multiprocesseur.

Nous ne nous intéressons pas aux détails techniques de l'application tels que les algorithmes de mise à l'échelle de compression. Ces derniers seront considérés comme des boîtes noires qui peuvent être remplacées par des algorithmes adéquats selon le choix et les besoins des utilisateurs. Ils ne font pas donc partie de notre étude.

9.2 Description des fonctionnalités JPEG

Les applications de traitement d'images et de vidéos sont de plus en plus présentes dans notre vie quotidienne. La particularité de ces applications est qu'elles opèrent sur de grandes masses de données et font souvent appels à des processus de calcul parallèle à haute performance. Ses applications contiennent dans leur description des contraintes temporelles telles que les périodicités et les précédences.

Nous présentons, par la suite, l'algorithme JPEG, constitué de plusieurs fonctions typiques de traitement de signal. Cet algorithme est dédié à la compression d'images.

Les principes de l'encodage

JPEG représente un standard dans le domaine de compressions d'images ayant une perte à taux variable (voir Figure 9.1). Il a connu un grand succès grâce à ses capacités et sa maniabilité. Par exemple, la plupart des images, se trouvant sur internet, sont compressées avec ce format. L'algorithme de base divise l'image source en des tableaux de taille (8×8) pixels. Ces derniers sont traités séparément selon la schéma illustré dans la Figure 9.2.

Ce n'est pas l'objectif de cette thèse de discuter de l'efficacité, des performances et des limitations de ce format. Pour cela, nous présentons brièvement le mode de fonctionnement de l'encodeur JPEG[1].

Figure 9.1: Trois exemple d'une photo compressée en JPEG, avec des compressions de plus en plus fortes, de gauche à droite [130].

L'algorithme d'encodage est composé de sept étapes illustrées dans la Figure 9.2. D'abord, on commence par découper l'image en blocs de 64 pixels (8×8). La JPEG est capable de coder les couleurs sous n'importe quel format. Toutefois les meilleurs taux de compression sont obtenus avec des codages de couleurs de types luminance/chrominance tels que YUV, YCbCr. Cela s'explique par le fait que l'œil est assez sensible à la luminance mais peu à la chrominance. Pour cela, une transformation de couleur est appliquée sur les blocs de pixels pour extraire les informations concernant la luminance et la chrominance. Ensuite, une transformation DCT est appliquée à chaque bloc de pixels. Cette transformation permet d'exprimer les informations de l'image en terme

[1]Pour plus de détails sur l'encodeur JPEG, le lecteur peut se référer à une description plus formelle dans [127].

de fréquence et d'amplitude plutôt qu'en pixels et couleurs. Une fois les fréquences et les amplitudes extraites, une étape de quantification est appliquée sur le résultat. La quantification représente la base de la compression car les informations peuvent être perdus durant cette étape. La matrice retournée par la DCT, de taille (8×8), est divisée par une autre matrice (appelé matrice de quantification) permettant ainsi d'atténuer la fréquence. Ensuite, un codage du type *Huffman* est appliqué sur la matrice de sortie. Le résultat de ce codage amène enfin à une image compressée.

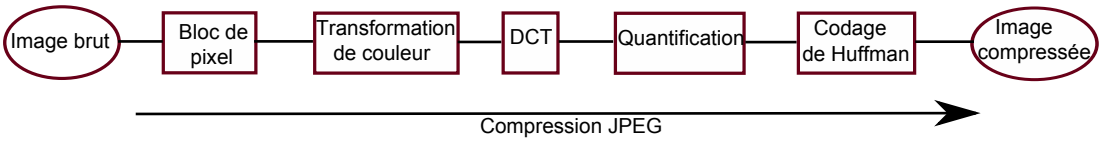


Figure 9.2: Organigramme de compression JPEG.

Maintenant que nous avons décris l'algorithme de compression JPEG, nous passons à la spécification des fonctionnalités de cet algorithme en utilisant UML/ Marte comme langage de modélisation de haut niveau.

9.3 Modélisation du comportement fonctionnel

Dans le cadre du projet Gaspard2, la spécification des différentes parties d'un système (application, architecture, association et déploiement) se font dans l'environnement de développement Eclipse. Gaspard2 bénéficie de l'intégrabilité de l'éditeur Papyrus dans l'environnement Eclipse. Cet éditeur utilise UML comme langage de modélisation primaire et permet l'ajout de profils afin d'enrichir la modélisation avec des détails spécifiques au domaine de conception. Ainsi, toutes les approches de développement en partant de la modélisation, en passant par les techniques de transformation de modèles jusqu'à la génération automatique de code, sont structurées autour de la plateforme Eclipse.

La Figure 9.3 montre une modélisation de l'encodeur JPEG, à un haut niveau d'abstraction, pour une image de taille (256×256). Nous utilisons le langage UML pour créer les composants englobants, les ports, les connecteurs ainsi que les instances de composants contenues dans le composant englobant. De plus, nous nous servons du profil Marte pour ajouter plus de détails relatifs à la spécification, notamment la description de parallélismes de données. Les ports d'entrées sont de types `FlowPort` (un stéréotype de Marte) ayant la direction `in`. De la même façon, les ports de sorties sont stéréotypés `FlowPort` avec une direction `out`. Pour des raisons de lisibilité, ces stéréotypes sont volontairement omis dans la Figure 9.3.

Expression du parallélisme de données

Nous utilisons le paquetage RSM afin d'exprimer le parallélisme de données d'une application manipulant des structures de données multi-dimensionnelles. Dans le haut de la Figure 9.3, le composant `JPEG Encoder` correspond au plus haut niveau hiérarchique de la description de composants. Le parallélisme de données est représenté par la répétition de l'instance `I` du composant `JpegComponent`. En effet, le stéréotype `Shaped`, ayant un vecteur de valeur {64,16}, exprime l'espace de répétition du composant `JpegComponent`. Cela signifie que ce composant est répété (64×16) fois. Chacune des répétitions consomme des motifs en entrée, de taille (8×8), qui sont extraits des tableaux

d'entrée du composant englobant `JPEG Encoder`. De même, chacune des répétitions produit des motifs en sortie, aussi de taille (8×8). Ces motifs sont consommés par les tableaux en sortie du composant englobant. L'attribut `fitting` du type matrice permet de déterminer les éléments du tableau qui sont liés à chaque motif. La matrice `paving` décrit l'espacement régulier des différents motifs au sein d'un tableau d'entrée ou de sortie. En d'autres termes, elle permet d'identifier l'origine de chaque motif, associé à chaque répétition de composant. Le vecteur `origin` spécifie l'origine du motif initial dans un tableau.

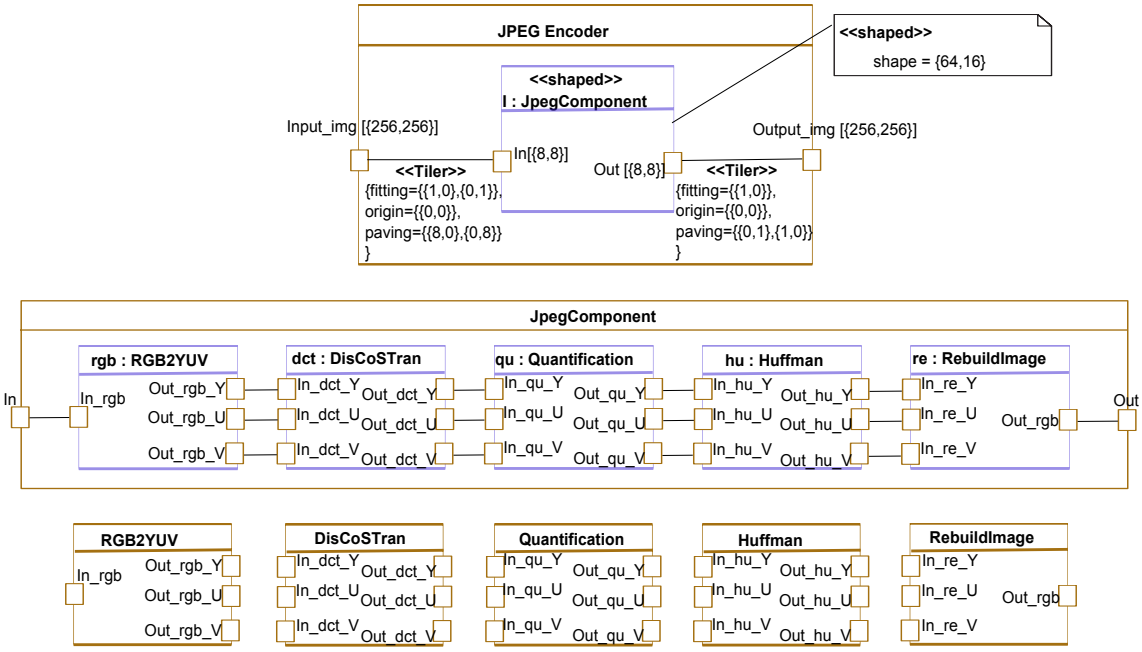


Figure 9.3: Modélisation Marte en Papyrus de l'algorithme de compression/décompression JPEG.

Expression du parallélisme de tâches

L'algorithme de codage d'images JPEG est représenté par le composant `JpegComponent`, illustré au milieu de la Figure 9.3. Il contient cinq instances de composants élémentaires. Un bloc de pixel de taille (8×8) est consommé par l'instance `rgb` du composant `RGB2YUV`. Ce dernier transforme l'information de couleur d'une image en luminance et chrominance. Ensuite, l'instance `dct` du composant `DisCoSTran` est appliquée sur le résultat suivie d'une application de l'instance `qu` du composant `Quantification`. Après l'étape de quantification, le résultat est codé avec l'instance `hu` du composant `Huffman`. L'image est enfin reconstituée avec l'instance `re` du composant `RebuildImage`.

Le bas de la Figure 9.3 montre cinq composants élémentaires `RGB2YUV`, `DisCoSTran`, `Quantification`, `Huffman` et `RebuildImage`. Ils représentent des composants élémentaires considérés comme étant des boîtes noires. Ce type de composants est lié, dans une phase de déploiement, à une bibliothèque d'IPs.

Nous considérons par la suite une description de contraintes de précédences imposées par construction du modèle fonctionnel. Ses contraintes sont décrites à l'aide d'horloges fonctionnelles, dont la sémantique est définie dans le chapitre 5.

Description de contraintes de dépendances de tâches

Nous analysons le graphe de dépendance de tâches au niveau application. Cette analyse consiste à trouver les composants qui sont dépendants l'un de l'autre et ceux qui sont totalement indépendants. Ceci nous permet de déduire des contraintes fonctionnelles, imposées par construction de modèles au niveau application. Ces contraintes représentent un ordre d'exécution de tâches fonctionnelles. Cet ordre doit être vérifié après l'étape d'association[2].

L'analyse repose essentiellement sur les composants de types élémentaires. Par exemple, dans la Figure 9.3, la première instance `rgb` du composant `RGB2YUV` ne peut être activée avant que le port d'entrée `in_rgb` du composant `JpegComp` soit plein. De la même manière, une activation de l'instance `dct` du composant `DisCoSTran` ne peut avoir lieu avant que les trois ports de sorties `out_rgb_Y`, `out_rgb_U` et `out_rgb_V` de l'instance `rgb` ne soient pleins.

Modèle d'exécution parallèle : dans le but d'exprimer le niveau de parallélisme maximal de tâches, nous considérons le modèle d'exécution parallèle, définis dans le chapitre 5. Ce modèle associe à chaque répétition d'une instance de composant élémentaire une horloge abstraite fonctionnelle. La Figure 9.4 montre plusieurs traces d'horloges fonctionnelles décrivant des contraintes de dépendances de tâches sur l'application JPEG. Dans chacune de ces traces, il existe cinq horloges fonctionnelles. En effet, pour chaque répétition d'une instance `I`, cinq horloges clk_{1_r}, clk_{2_r}, clk_{3_r}, clk_{4_r} et clk_{5_r} sont créées et associées respectivement aux instances `rgb`, `dct`, `qu`, `hu` et `re`, où r dénote le numéro d'une répétition. Il existe en tout n traces d'horloges fonctionnelles et $n \times 5$ horloges fonctionnelles, où $n = 16 \times 64 = 1024$ représente l'espace de répétition de l'instance `I` du composant `JpegComponent`. Cette distribution d'horloges conserve le niveau maximal de parallélisme de tâches dans l'application JPEG modélisée en Marte.

Dans la Figure 9.4, la première trace d'horloges abstraites illustre les dépendances de tâches durant l'exécution de la première répétition de l'instance `I`. L'horloge clk_{1_1} est associée à la première instance `rgb` du composant `RGB2YUV` contenue dans le composant `JpegComp`. Cette horloge décrit une dépendance de tâche avec l'horloge fonctionnelle clk_{2_1} associée à l'instance de composant `dct`. Cette dépendance est strictement liée au même niveau de répétition. En d'autre terme, aucune dépendance n'existe entre les différentes traces d'horloges reflétant les différentes niveaux de répétitions de l'instance `I`. Donc des instances dans différents niveaux peuvent être exécuter en parallèle.

clk_{1_1}:	1	0	0	0	0	clk_{1_2}:	1	0	0	0	0	...	clk_{1_n}:	1	0	0	0	0
clk_{2_1}:	0	1	0	0	0	clk_{2_2}:	0	1	0	0	0	...	clk_{2_n}:	0	1	0	0	0
clk_{3_1}:	0	0	1	0	0	clk_{3_2}:	0	0	1	0	0	...	clk_{3_n}:	0	0	1	0	0
clk_{4_1}:	0	0	0	1	0	clk_{4_2}:	0	0	0	1	0	...	clk_{4_n}:	0	0	0	1	0
clk_{5_1}:	0	0	0	0	1	clk_{5_2}:	0	0	0	0	1	...	clk_{5_n}:	0	0	0	0	1

Figure 9.4: Trace des horloges générer pour chaque répétition de la tâche JpegComp d'une image de taille (256×256), selon le modèle d'exécution parallèle.

Modèle d'exécution pipeline : le modèle d'exécution parallèle, décris ci-dessus, considère un niveau de parallélisme maximal. Ceci dit, chaque instance de composant est

[2]L'analyse de dépendances de tâches est effectuée avant l'étape d'association donc sans connaître a priori les particularités de l'architecture cible ni la répartition des fonctionnalités sur les différentes unités de calcul.

exécutée par un processeur virtuel. Or, dans une implémentation réelle, un nombre limité de processeur est considéré. Donc l'exécution de toutes les instances de tâches sur des processeurs distincts n'est pas un choix de configuration réaliste, surtout quand le modèle fonctionnel contient des milliers d'instances de composants élémentaires.

Nous définissons ainsi une nouvelle trace d'horloges abstraites en choisissant le modèle d'exécution pipeline qui réduit le niveau de parallélisme de tâches. Nous abstrayons ainsi l'application JPEG par une trace de cinq horloges fonctionnelles clk_1, clk_2, clk_3, clk_4 et clk_5 illustrées dans la Figure 9.5. Ces horloges sont associées respectivement aux instances `rgb`, `dct`, `qu`, `hu` et `re` respectivement associées aux composants `RGB2YUV`, `DisCoSTran`, `Quantification`, `Huffman` et `RebuildImage`. L'activation d'une instance `rgb` (associée à l'horloge clk_1) n'est pas contraignante car les données sont présentes à tout moments. Cependant, l'instance `dct` par exemple (associée à l'horloge clk_2) ne peut être activée tant que l'instance `rgb` n'a pas était activée. Cette dépendance de tâche est évidente car le composant `RGB2YUV` a le rôle d'un producteur et le composant `DisCosTran` a le rôle d'un consommateur.

$$
\begin{array}{lcccccccccccccc}
clk_1\colon & 1 & 1 & 1 & 1 & 1 & 1 & 1 & \ldots & 1 & 1 & 0 & 0 & 0 & 0 \\
clk_2\colon & 0 & 1 & 1 & 1 & 1 & 1 & 1 & \ldots & 1 & 1 & 1 & 0 & 0 & 0 \\
clk_3\colon & 0 & 0 & 1 & 1 & 1 & 1 & 1 & \ldots & 1 & 1 & 1 & 1 & 0 & 0 \\
clk_4\colon & 0 & 0 & 0 & 1 & 1 & 1 & 1 & \ldots & 1 & 1 & 1 & 1 & 1 & 0 \\
clk_5\colon & 0 & 0 & 0 & 0 & 1 & 1 & 1 & \ldots & 1 & 1 & 1 & 1 & 1 & 1
\end{array}
$$

Figure 9.5: Trace de cinq horloges abstraites, abstrayant l'application JPEG d'une image (256×256), selon le modèle d'exécution pipeline.

La Figure 9.6 montre la même trace d'horloges spécifiée dans la Figure 9.5 mais avec une notation périodique.

$$
\begin{array}{lccccc}
clk_1\colon & (1)^{1024} & 0 & 0 & 0 & 0 \\
clk_2\colon & 0 & (1)^{1024} & 0 & 0 & 0 \\
clk_3\colon & 0 & 0 & (1)^{1024} & 0 & 0 \\
clk_4\colon & 0 & 0 & 0 & (1)^{1024} & 0 \\
clk_5\colon & 0 & 0 & 0 & 0 & (1)^{1024}
\end{array}
$$

Figure 9.6: Trace périodique de cinq horloges abstraites, abstrayant l'application JPEG d'une image (256×256), selon le modèle d'exécution pipeline.

9.4 Modélisation d'une architecture PRAM

Les architectures modélisées en Gaspard2 ont l'avantage d'être réutilisables. De plus, elles sont facilement extensibles. Lorsqu'une nouvelle application apparaît, ayant des charges de travails plus complexes, il suffit d'incrémenter le nombre de processeurs dans l'architecture modélisée pour augmenter la puissance de calcul.

Le but de cette section est de modéliser une architecture matérielle contenant un ensemble de processeurs homogènes dédiés à l'exécution d'applications hautes performances.

Description de l'architecture

Nous considérons dans cette section une architecture du type PRAM[3]. Les différents processeurs communiquent à travers une mémoire partagée. Chaque processeur est considéré comme étant une machine, à accès aléatoire, ayant des droits d'écriture et de lecture concurrents sur la mémoire. L'accès direct à la mémoire permet de stocker et de récupérer de données.

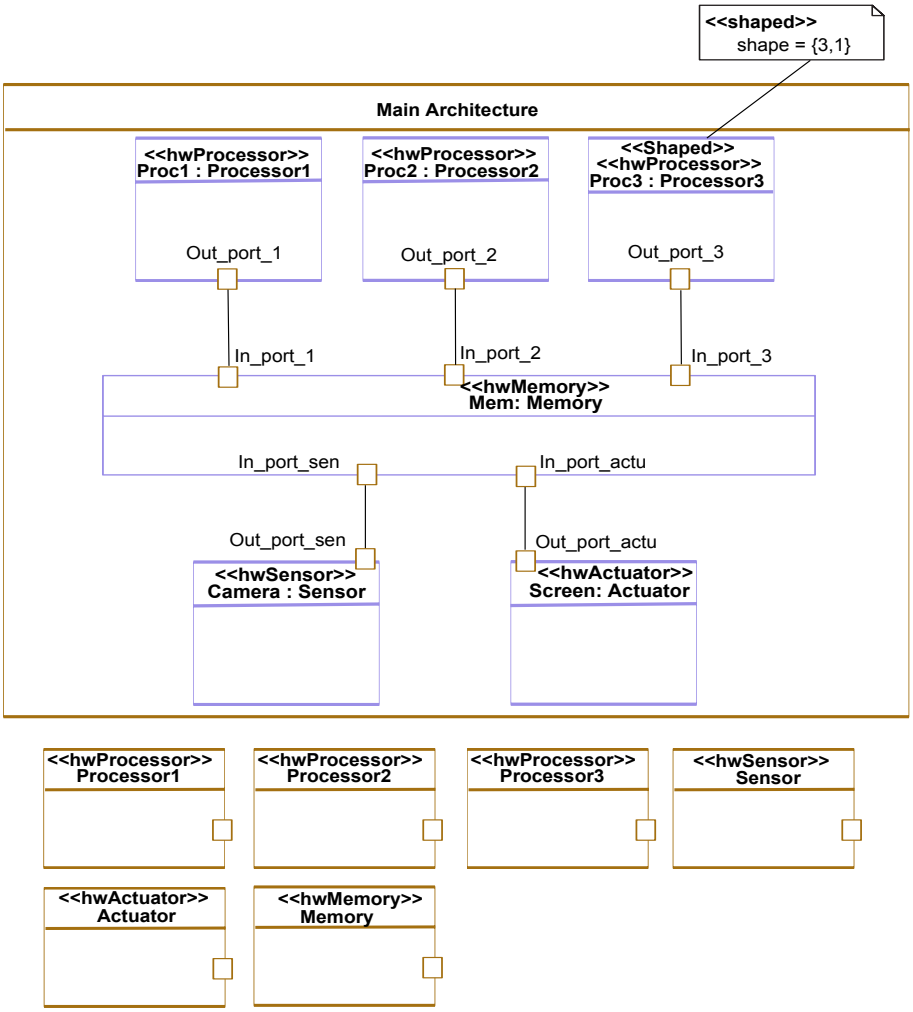


Figure 9.7: Modélisation Marte en Papyrus d'une architecture PRAM.

Modélisation de l'architecture

La Figure 9.7 montre un modèle d'architecture PRAM. Cinq processeurs sont définis pour traiter les différents composants fonctionnels de la JPEG. `Proc1` et `Proc2` sont définis comme étant des instances de processeurs unitaires alors que `Proc3` est une instance d'un processeur répétée trois fois. La répétition du processeur est exprimée via le stéréotype multi-dimensionnel `Shaped` ayant la valeur $\{3, 1\}$. Tout les processeurs sont stéréotypés `hwProcessor`. Un émetteur (l'instance de composant `Camera`) et un récepteur (l'instance de composant `Screen`) ont pour rôle de produire et de consommer des pixels. Ils sont respectivement stéréotypés `hwSensor` et `hwActuator` de Marte.

[3]Pour plus d'information sur l'architecture PRAM, le lecteur peut se référer au chapitre 6.

Nous définissons également une instance de mémoire partagée `Mem` (mémoire centrale partagée) pour écrire, stocker et récupérer les données et les instructions du programme. Toutes les instances de composants, contenues dans le composant `MainArchitecture`, sont stéréotypées avec des concepts du profil Marte (tous les stéréotypes commençant par `hw`).

Abstraction de l'architecture par le biais d'horloges abstraites physiques

Durant la modélisation de la architecture, le processeur `Proc1` (resp. `Proc2` et `Proc3`) lui est attribué une fréquence égale à 100 MHz (resp. 200 MHz et 300 MHz). Nous définissons $P = 1/f$ comme étant la période entre deux tops d'horloge d'un processeur, où f représente la fréquence du processeur.

En se basant sur les périodes des cycles d'horloges des différents processeurs, nous repérons les instants d'activation de chaque processeur. Les valeurs de fréquence ci-dessous sont extraites de la spécification de l'architecture décrite dans la Figure 9.7 qui implémente l'algorithme de codage JPEG :

- `Proc1` : f_1 = 100 MHz, $Period_1$ = 0.01 microsecondes
- `Proc2` : f_2 = 200 MHz, $Period_2$ = 0.005 microsecondes
- `Proc3` : f_3 = 300 MHz, $Period_3$ = 0.0033 microsecondes

À partir des périodes de processeurs calculées ci-dessus, nous définissons trois horloges abstraites physiques $PhysicalClk_1$, $PhysicalClk_2$ et $PhysicalClk_3$ associées respectivement aux processeurs $Proc1$, $Proc2$ et $Proc3$. Nous définissons aussi une horloge, appelée *IdealClk*, qui synchronise les instants des différentes horloges physiques sur ses propres instants. Afin de caractériser *IdealClk*, nous calculons sa période $P_{IdealClk} = 1/PPCM(f_1, ..., f_n)$, où $PPCM$ dénote le plus petit commun multiple et n dénote le nombre maximal de processeurs impliqués dans l'exécution. Pour les fréquences ci-dessus, nous obtenons $PPCM(100, 200, 300) = 600$ et $P_{IdealClk} = 0.0016$ microsecondes.

Figure 9.8: Horloges physiques associées à des processeurs de types RAM et cadencées sur une horloge de référence.

La Figure 9.8 illustre la synchronisation des cycles d'horloges des différents processeurs sur les cycles de l'horloge de référence *IdealClk*.

9.5 Modélisation d'associations

L'association dans Gaspard2 consiste à attribuer pour chaque composant fonctionnel une ressource physique correspondante qui va prendre en charge son exécution. Nous allouons par exemple des ports d'entrées/sorties à des mémoires et des composants fonctionnels à des unités de calculs.

Quand des processeurs sont bien employés, ils peuvent réduire considérablement la consommation d'énergie d'un processeur. Il est vrai que la durée de traitement en sera probablement affectée, mais le plus important est que le système, dans son ensemble,

Exemples	Processeurs	Type d'association	Répartition des horloges
1	Proc1	tâches/processeurs	$clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$
2	Proc1 Proc2	tâches/processeurs	clk'_1, clk'_2, clk'_3 clk'_4, clk'_5
3	Proc1 Proc2 Proc3	tâches/processeurs	clk'_1 clk'_2 clk'_3, clk'_4, clk'_5
4	Proc1 Proc2 Proc3 Proc4	tâches/processeurs	clk'_1 clk'_2 clk'_3 clk'_4, clk'_5
5	Proc1 Proc2 Proc3 Proc4 Proc5	tâches/processeurs	clk'_1 clk'_2 clk'_3 clk'_4 clk'_5
6	Proc1 Proc2	sous-images/processeurs	$clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$ $clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$
7	Proc1 Proc2 Proc3	sous-images/processeurs	$clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$ $clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$ $clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$
8	Proc1 Proc2 Proc3 Proc4	sous-images/processeurs	$clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$ $clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$ $clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$ $clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$
9	Proc1 Proc2 Proc3 Proc4 Proc5	sous-images/processeurs	$clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$ $clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$ $clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$ $clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$ $clk'_1, clk'_2, clk'_3, clk'_4, clk'_5$

Table 9.1: Neuf exemples de configurations d'une architecture et d'une associations.

respecte les contraintes temporelles et les temps d'échéances imposés sur l'application. Il s'agit donc, comme toujours, de trouver le juste équilibre.

Ainsi, pour une architecture MPSoC, plusieurs choix d'associations peuvent être considérés. Ça revient au concepteur de choisir la configuration la plus adéquate. Par exemple, les Figures 9.9, 9.10 et 9.11 représentent respectivement des associations de un, deux et cinq processeurs pour l'exécution des différentes tâches élémentaires de l'encodeur JPEG. Dans le même contexte, les Figures 9.12 et 9.13 représentent des associations de deux et de cinq processeurs respectivement à deux et cinq sous-images. Le Tableau 9.1 montre des exemples de neuf configurations possibles des architectures et des associations. Le exemples 1, 2, 3, 4 et 5 décrivent des associations de différentes tâches sur différents processeurs. Les exemples 6, 7, 8 et 9 représentent, à leur tour, différentes types d'associations de sous-images sur différents processeurs.

Exemple d'association mono-processeur

La Figure 9.9 montre une association possible d'une application JPEG sur une architecture PRAM. Ici, seulement le processeur `Proc2` est impliqué dans l'exécution des

fonctionnalités, représentées par le composant `JpegComponent`. Afin de décrire ce système avec des horloges abstraites, l'algorithme d'ordonnancement multi-tâches/mono-processeur doit être appliqué.

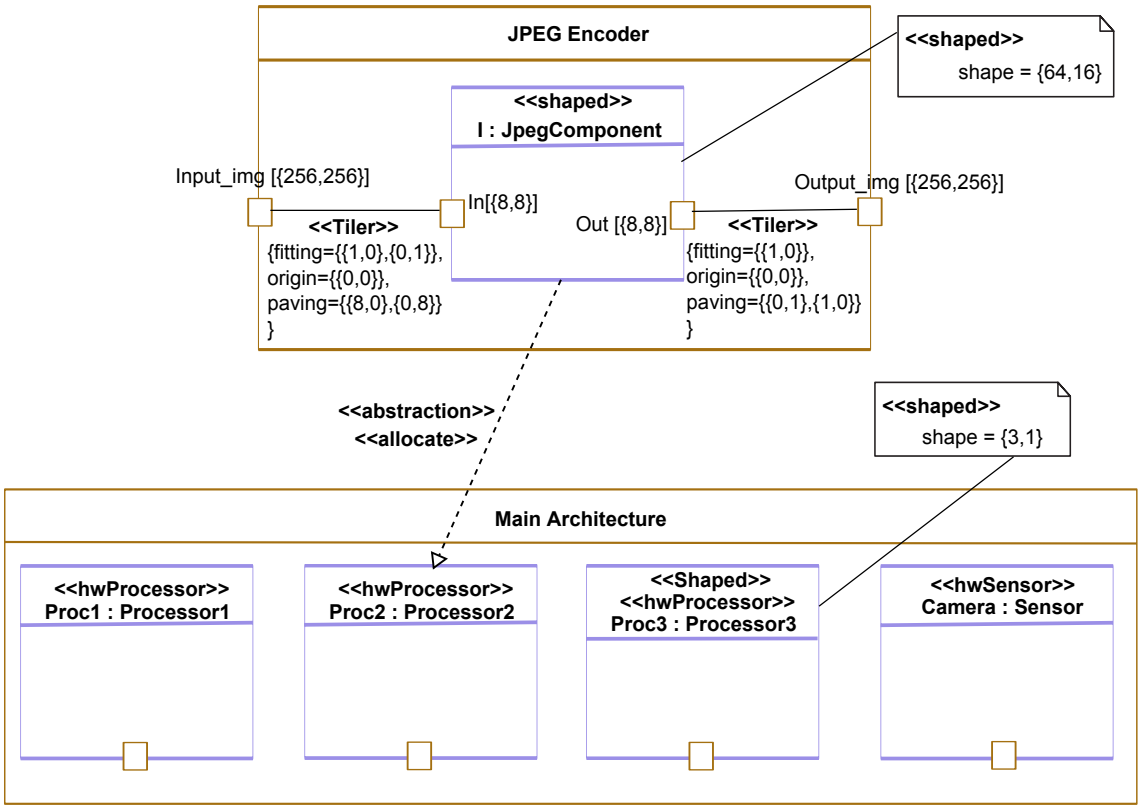


Figure 9.9: Modélisation Marte d'une association entre les composants fonctionnels et un processeur (**exemple 1**).

Exemples d'associations de composants élémentaires sur des processeurs

Dans la Figure 9.10, nous considérons les deux processeurs `Proc1` et `Proc2` dans l'exécution des fonctionnalités. `Proc1` est alloué aux instances `rgb`, `dct` et `qu` alors que `Proc2` est alloué aux instances `hu` et `re`. Afin d'analyser le système à l'aide d'horloges abstraites, nous considérons un algorithme d'ordonnancement multi-tâches/multiprocesseur.

Dans la Figure 9.11, cinq processeurs sont considérés dans l'exécution de l'encodeur JPEG. Les deux processeurs `Proc1` et `Proc2` sont unitaires alors que le processeur `Proc3` est du type répétitif. Ce dernier est répété 3 fois selon la valeur du stéréotype `Shaped` ayant la valeur $\{3, 1\}$. Le résultat de cette répétition est la définition de trois sous processeurs $\texttt{Proc3}_1$, $\texttt{Proc3}_2$ et $\texttt{Proc3}_3$ ayant les mêmes caractéristiques que `Proc3`. Les instances `rgb` et `dct` sont associées respectivement aux processeurs `Proc1` et `Proc2` à travers des connecteurs stéréotypés `allocate`. Les instances `qu`, `hu` et re sont associées respectivement aux processeurs $\texttt{Proc3}_1$, $\texttt{Proc3}_2$ et $\texttt{Proc3}_3$ à travers un connecteur de type `distribute`.

Pour décrire cette association avec des horloges abstraites, il suffit d'appliquer l'algorithme d'ordonnancement mono-tâche/mono-processeur pour chacun des cinq processeurs impliqués dans l'exécution (chaque processeur étant alloué une tâche fonctionnelle).

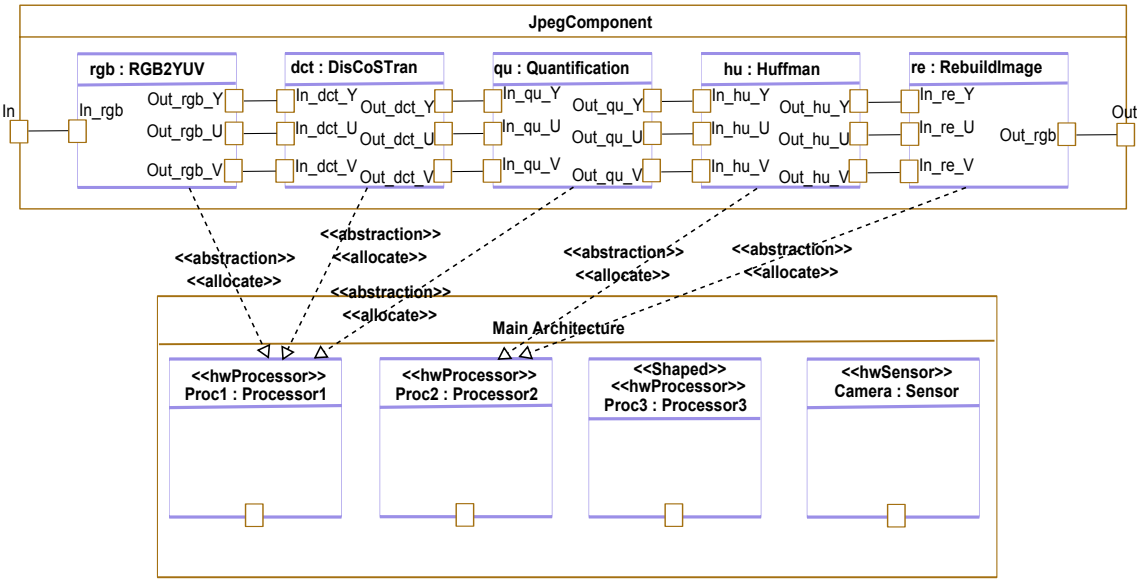


Figure 9.10: Modélisation Marte d'une association entre les composants fonctionnels et deux processeurs (**exemple 2**).

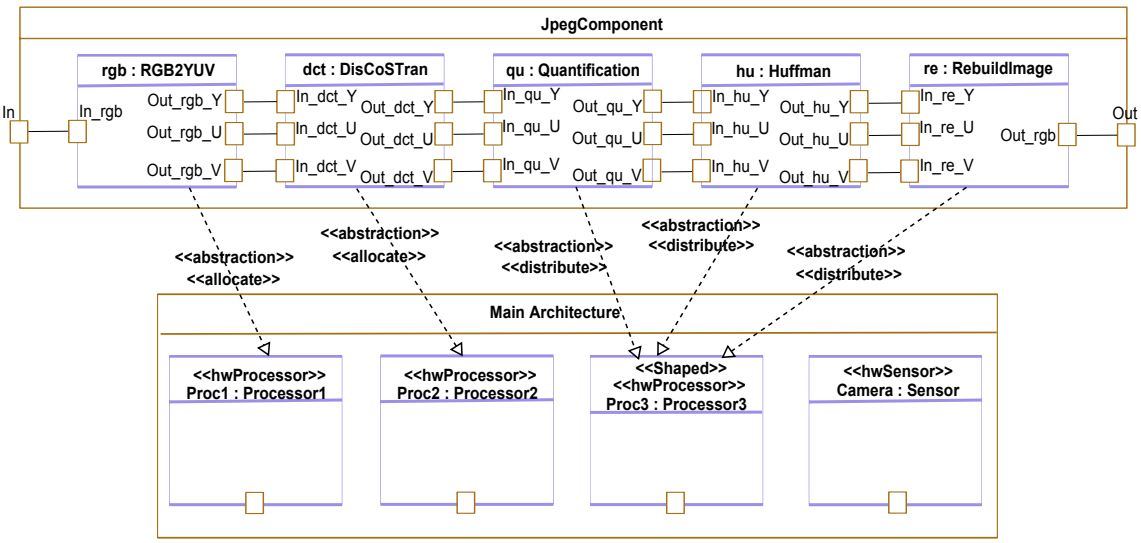


Figure 9.11: Modélisation Marte d'une association entre les composants fonctionnels et cinq processeurs (**exemple 5**).

Exemples d'associations de parties d'une image sur des processeurs

Nous montrons des exemples d'associations de parties d'images sur des processeurs distincts. La Figure 9.12 par exemple montre une association d'une image sur deux processeurs `Proc1` et `Proc2`; l'image étant divisée en deux parties égales que nous appelons sous-parties.

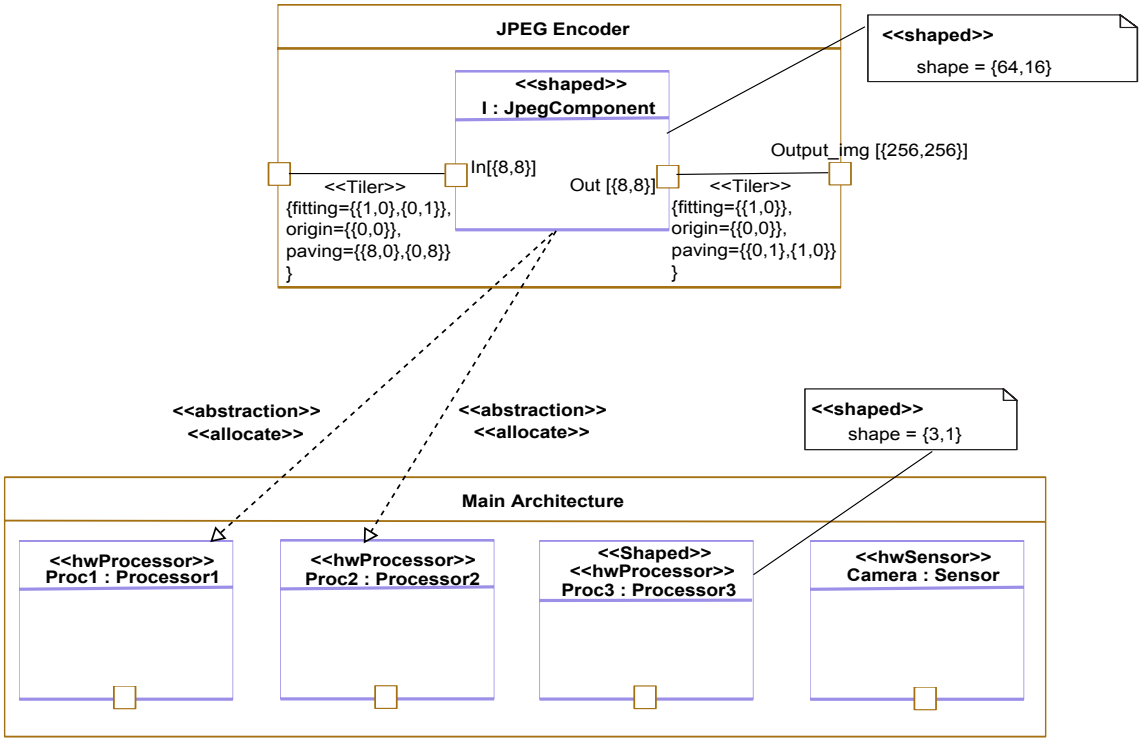


Figure 9.12: Modélisation Marte d'une association en partitionnant l'image en deux (**exemple 6**).

De la même manière, la Figure 9.13 montre une association d'une image sur cinq processeurs `Proc1`, `Proc2`, $\texttt{Proc3}_1$, $\texttt{Proc3}_2$ et $\texttt{Proc3}_3$; l'image d'entrée étant divisée en cinq parties égales. Afin d'analyser, par le biais d'horloges abstraites, les configurations de l'architecture et de l'association illustrées par les Figures 9.12 et 9.13, nous appliquons l'algorithme d'ordonnancement multi-tâches/multiprocesseur définis dans le chapitre 7.

9.6 Synthèse de propriétés

Dans cette section, plusieurs modèles d'une architecture et d'une association sont analysés par le biais d'horloges abstraites. L'objectif final est de guider le concepteur dans le choix des configurations de l'architecture et de l'association les plus adéquates par rapport aux besoins. Parmi les différentes configurations possibles, le concepteur doit choisir :

- le nombre de processeurs idéal pour l'exécution de fonctionnalités ;
- la valeur de fréquence/tension de chaque processeur ;
- le type d'association de tâches sur des processeurs ;

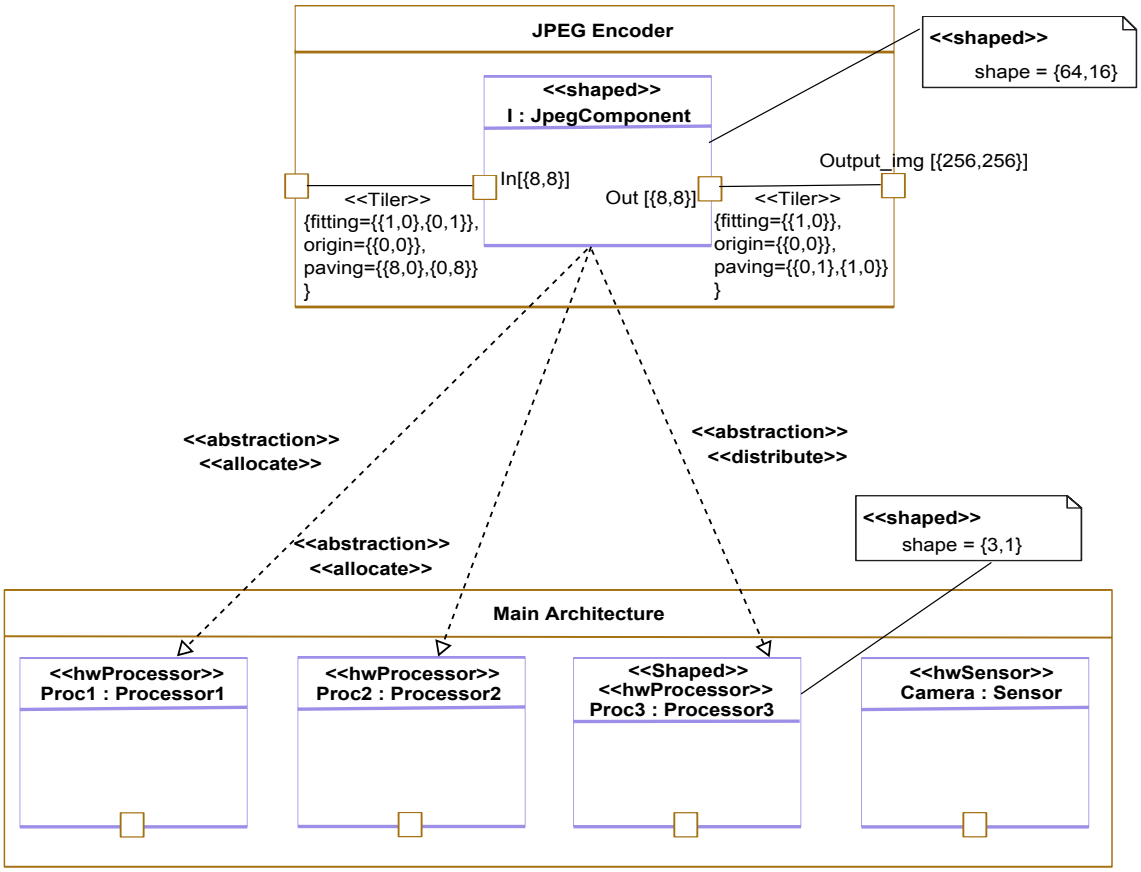


Figure 9.13: Modélisation Marte d'une association en partitionnant l'image en cinq (**exemple 9**).

Les critères ci-dessus affectent énormément les performances de systèmes. À chaque modification d'un de ces critères, les informations suivantes doivent être réévaluées :

- le respect des contraintes de dépendances de tâches ;
- le temps d'exécution des différentes tâches fonctionnelles ;
- la consommation d'énergie totale du système.

9.6.1 Expansion d'horloges abstraites fonctionnelles

Dans l'étude de cas de l'encodeur JPEG, nous considérons le modèle d'exécution pipeline pour décrire les fonctionnalités du système. Les horloges abstraites fonctionnelles résultantes de ce modèle d'exécution sont illustrées dans la Figure 9.14.

clk_1:	1	1	1	1	1	1	1	...	1	1	0	0	0	0
clk_2:	0	1	1	1	1	1	1	...	1	1	1	0	0	0
clk_3:	0	0	1	1	1	1	1	...	1	1	1	1	0	0
clk_4:	0	0	0	1	1	1	1	...	1	1	1	1	1	0
clk_5:	0	0	0	0	1	1	1	...	1	1	1	1	1	1

Figure 9.14: Trace de cinq horloges abstraites fonctionnelles clk_1, clk_2, clk_3, clk_4 et clk_5 représentant l'algorithme de codage JPEG pour une image de taille (256×256) et selon une exécution pipeline.

Cinq horloges abstraites clk_1, clk_2, clk_3, clk_4 et clk_5 sont identifiées dans cette trace. Elles représentent respectivement les activations des instances de tâches `rgb`, `dct`, `qu`, `hu` et `re`. Chacune de ces horloges contient des valeurs binaires 0 ou 1. Le nombre d'occurrence de la valeur 1 est de $64 \times 16 = 1024$ reflétant le nombre d'activation des instances allouées pour chaque horloge.

Nombre de cycles horloges pour les tâches élémentaires

Pour synthétiser une trace d'horloges ternaires, simulant *l'activité des processeurs*, nous devons d'abord étendre les horloges abstraites fonctionnelles. L'expansion de ces horloges revient à remplacer les occurrences de la valeur 1 par le nombre de cycles d'horloge processeur nécessaire à l'exécution de la tâche concernée. Cette exécution inclue la lecture de données, le traitement et le sauvegarde dans la mémoire. L'information concernant le nombre de cycles d'horloge dépend de la complexité de la tâche fonctionnelle ainsi que le type du processeur qui l'exécute[4].

Pour chaque tâche élémentaire T_i, le concepteur doit donc indiquer le nombre de cycles d'horloge processeur c_i nécessaire pour son exécution[5].

Le Tableau 9.2 montre une estimation du nombre de cycles d'horloge nécessaire pour exécuter une répétition des instances `rgb`, `dct`, `qu`, `hu` et `re`. L'exécution des instances comporte la consommation, le traitement et la génération de blocs de pixels de taille (8×8) pixels. Ces mesures sont obtenues durant l'exécution de l'application JPEG sur un processeur RISC du type `PowerPC 405 CPU Core` en fixant sa fréquence à 300 MHz.

[4]Le lecteur peut se référer au chapitre 7 pour plus d'informations sur l'expansion d'horloges.

[5]Nous pouvons imaginer une bibliothèque d'IP logiciel contenant des informations de cycles d'horloge pour chaque composant élémentaire fonctionnel.

Instance	Répétition	Horloge binaire	Cycles processeur
rgb	1024	clk_1	c_1 =7534
dct	1024	clk_2	c_2 =7214
qt	1024	clk_3	c_3 =6920
hu	1024	clk_4	c_4 =5504
re	1024	clk_5	c_5 =4594

Table 9.2: Cycles d'horloge processeur nécessaire pour exécuter une répétition des instances `rgb`, `dct`, `qu`, `hu` et `re`.

Expansion d'horloges fonctionnelles

Nous découpons les horloges abstraites fonctionnelles en des tranches de taille 64. Chaque tranche contient une suite de valeurs ternaires 0, 1 ou -1. Le choix de la taille des tranches dépend des différents types d'ordonnancement souhaités.

La Figure 9.15 montre une trace de cinq horloges étendues clk'_1, clk'_2, clk'_3, clk'_4 et clk'_5. Ces horloges représentent l'expansion des horloges fonctionnelles, découpées en tranches de même taille.

$$
\begin{aligned}
clk'_1 &: [1\ \ (-1)^{c_1-1}]^{64} \quad [[1\ \ (-1)^{c_1-1}]^{64}]^{15} \quad 0\ 0\ 0\ 0 \\
clk'_2 &: 0\ [1\ \ (-1)^{c_2-1}]^{63} \quad [[1\ \ (-1)^{c_2-1}]^{64}]^{15} \quad [1\ \ (-1)^{c_2-1}0\ 0\ 0 \\
clk'_3 &: 0\ 0\ [1\ \ (-1)^{c_3-1}]^{62} \quad [[1\ \ (-1)^{c_3-1}]^{64}]^{15} \quad [1\ \ (-1)^{c_3-1}]^2 0\ 0 \\
clk'_4 &: 0\ 0\ 0\ [1\ \ (-1)^{c_4-1}]^{61} \quad [[1\ \ (-1)^{c_4-1}]^{64}]^{15} \quad [1\ \ (-1)^{c_4-1}]^3 0 \\
clk'_5 &: 0\ 0\ 0\ 0\ [1\ \ (-1)^{c_5-1}]^{60} \quad [[1\ \ (-1)^{c_5-1}]^{64}]^{15} \quad [1\ \ (-1)^{c_5-1}]^4
\end{aligned}
$$

Figure 9.15: Trace de cinq horloges binaires clk'_1, clk'_2, clk'_3, clk'_4 et clk'_5 représentant l'expansion des horloges clk_1, clk_2, clk_3, clk_4, et clk_5 selon les valeurs c_1, c_2, c_3, c_4 et c_5.

Les horloges étendues clk'_i, où $i \in [1, 5]$, sont divisées en dix sept tranches T_i^j, où i représente l'indice de l'horloge clk'_i et j représente l'indice de la $j^{\text{ième}}$ tranche dans clk_i. Ces horloges sont organisées comme suit :

- la première tranche T_i^1 de l'horloge clk'_i a la couleur rouge ;
- les quinze tranches T_i^j suivantes, où $j \in [2, 15]$ ont la couleur noir ;
- La dernière tranche T_i^{17} de l'horloge clk'_i a la couleur bleu.

Pour simplifier la lecture des différentes horloges étendues, d'une part, et pour tracer les horloges ternaires représentant l'activité des différents processeurs concernés dans l'exécution, d'autre part, nous écrivons les horloges étendues sous forme d'un ensemble de tranches T_i^j :

- clk'_1 :
  - $T_1^1 = [1\ \ (-1)^{c_1-1}]^{64}$
  - $T_1^j = [1\ \ (-1)^{c_1-1}]^{64}$, où $j \in [2, 16]$
  - $T_1^{17} = 0\ \ 0\ \ 0\ \ 0$
- clk'_2 :
  - $T_2^1 = 0\ \ [1\ \ (-1)^{c_2-1}]^{63}$
  - $T_2^j = [1\ \ (-1)^{c_2-1}]^{64}]$, où $j \in [2, 16]$
  - $T_2^{17} = 1\ \ (-1)^{c_2-1}0\ \ 0\ \ 0$

- clk'_3 :
  - $T_3^1 = 0\ 0\ [1\ \ (-1)^{c_3-1}]^{62}$
  - $T_3^j = [1\ \ (-1)^{c_3-1}]^{64}$, où $j \in [2, 16]$
  - $T_3^{17} = [1\ \ (-1)^{c_3-1}]^2 0\ \ 0$
- clk'_4 :
  - $T_4^1 = 0\ 0\ 0\ [1\ \ (-1)^{c_4-1}]^{61}$
  - $T_4^j = [1\ \ (-1)^{c_4-1}]^{64}]$, où $j \in [2, 16]$
  - $T_4^{17} = [1\ \ (-1)^{c_4-1}]^3 0$
- clk'_5 :
  - $T_5^1 = 0\ 0\ 0\ 0\ [1\ \ (-1)^{c_5-1}]^{60}$
  - $T_5^j = [1\ \ (-1)^{c_5-1}]^{64}]$, où $j \in [2, 16]$
  - $T_5^{17} = [1\ \ (-1)^{c_5-1}]^4$

La Figure 9.16 représente la nouvelle trace des horloges étendues clk'_1, clk'_2, clk'_3, clk'_4 et clk'_5, sous forme d'un ensemble ordonné de tranches. Nous considérons cette trace d'horloges pour montrer différents types d'ordonnancements de fonctionnalités JPEG sur des architectures PRAM. Ces ordonnancements sont représentés par le biais de projections d'horloges abstraites étendues sur des horloges physiques associées aux différents processeurs.

clk'_1 :	T_1^1	T_1^2	T_1^3	...	T_1^{16}	T_1^{17}
clk'_2 :	T_2^1	T_2^2	T_2^3	...	T_2^{16}	T_2^{17}
clk'_3 :	T_3^1	T_3^2	T_3^3	...	T_3^{16}	T_3^{17}
clk'_4 :	T_4^1	T_4^2	T_4^3	...	T_4^{16}	T_4^{17}
clk'_5 :	T_5^1	T_5^2	T_5^3	...	T_5^{16}	T_5^{17}

Figure 9.16: Trace, sous forme de tranches, des cinq horloges abstraites étendues clk'_1, clk'_2, clk'_3, clk'_4 et clk'_5.

9.6.2 Projection d'horloges abstraites fonctionnelles sur des horloges abstraites physiques

Dans cette section, nous considérons des projections d'horloges abstraites étendues, associées à des tâches fonctionnelles, sur des horloges physiques associées à des processeurs. Ces projections d'horloges représentent différentes types d'associations. Par exemple, nous considérons des associations de plusieurs tâches sur un seul processeur et de plusieurs tâches sur plusieurs processeurs. Aussi, nous considérons plusieurs cas d'associations de processeurs à des sous-images.

Ordonnancement de plusieurs tâches élémentaires sur un processeur

Nous considérons l'exécution des fonctionnalités JPEG sur un seul processeur. Dans ce cas, le processeur commute son exécution sur l'ensemble des tâches concernées jusqu'à la fin de l'exécution de toutes les tâches fonctionnelles.

Ce type d'association est modélisé en Marte dans l'exemple 1 de la Figure 9.9. Dans ce cas, toute l'application JPEG, contenue dans le composant `JpegComponent`, est allouée au processeur `Proc2`. Ce dernier commence son exécution par la tranche T_1^1, représentant la première tranche de l'horloge clk'_1. Ensuite, il exécute les premières tranches des horloges clk'_2, clk'_3, clk'_4 et clk'_5. De la même manière, `Proc2` exécute les tranches restantes des différentes horloges d'une manière récursive jusqu'à finir toutes les tranches des différentes horloges.

$$TPhysical_2 : T_1^1\ T_2^1\ T_3^1\ T_4^1\ T_5^1\ T_1^2\ T_2^2\ T_3^2\ T_4^2\ T_5^2\ T_1^3\ T_2^3\ T_3^3\ \ldots\ T_1^{17}\ T_2^{17}\ T_3^{17}\ T_4^{17}\ T_5^{17}$$

Figure 9.17: Trace de l'horloge abstraite $TPhysical_2$ simulant l'activité du processeur `Proc2`.

La Figure 9.17 montre une trace de l'horloge abstraite $TPhysical_2$, simulant l'activité du processeur `Proc2` durant l'encodage JPEG d'une image de taille (256×256). Le processeur exécute les différentes tranches d'une manière séquentielle en commencant par le première tranche T_1^1 et en finissant l'exécution par la tranche T_5^{17}.

Ordonnancement de plusieurs tâches élémentaires sur plusieurs processeurs

Nous considérons le cas d'associations de plusieurs tâches élémentaires, représentées par des horloges abstraites étendues, sur plusieurs processeurs. Dans ce genre d'ordonnancement, nous nous intéressons aux types de tâches, à leurs complexités et et leurs charges de travail. Nous allouons aux différents processeurs des tâches ayant des charges de travail presque égales. De cette manière, nous obtenons une distribution équitable de la charge de travail.

Dans la Figure 9.18, nous projetons les différentes tranches des horloges clk'_1, clk'_2 et clk'_3 sur les cycles d'horloge du processeur `Proc1`. De la même manière, nous projetons les différentes tranches des horloges clk'_4 et clk'_5 sur les cycles d'horloges du processeur `Proc2`. Les horloges abstraites $TPhysical_1$ et $TPhysical_2$ sont le résultat de cette projection. Ces dernières représentes l'activité des processeurs `Proc1` et `Proc2` durant l'exécution des tâches élémentaires. Cette association a été modélisée en Marte dans la Figure 9.10 (exemple 2).

$$TPhysical_1 : T_1^1\ T_2^1\ T_3^1\ T_1^2\ T_2^2\ T_3^2\ T_1^3\ T_2^3\ T_3^3\ \ldots\ T_1^{17}\ T_2^{17}\ T_3^{17}$$

$$TPhysical_2 : T_4^1\ T_5^1\ T_4^2\ T_5^2\ T_4^3\ T_5^3 \ldots\ T_4^{17}\ T_5^{17}$$

Figure 9.18: Trace des horloges abstraites $TPhysical_1$ et $TPhysical_2$ simulant respectivement l'activité des processeurs `Proc1` et `Proc2`.

La Figure 9.19 montre la trace de cinq horloges abstraites $TPhysical_1$, $TPhysical_2$, $TPhysical_3$, $TPhysical_4$ et $TPhysical_5$ simulant respectivement l'activité des processeurs `Proc1`, `Proc2`, $\texttt{Proc3}_1$, $\texttt{Proc3}_2$ et $\texttt{Proc3}_3$. Ce type d'association a été modélisé en Marte dans la Figure 9.11 (exemple 5). Chacun des processeurs est associé une tâche

fonctionnelle. Pour cela, chacune des horloges abstraites est projetée sur une horloge physique. Cela nous ramène au cas d'ordonnancement mono-tâche/mono-processeur.

$$TPhysical_1 : T_1^1\ T_1^2\ T_1^3\ T_1^4\ T_1^5\ T_1^6\ ...\ T_1^{16}\ T_1^{17}$$
$$TPhysical_2 : T_2^1\ T_2^2\ T_2^3\ T_2^4\ T_2^5\ T_2^6\ ...\ T_2^{16}\ T_2^{17}$$
$$TPhysical_3 : T_3^1\ T_3^2\ T_3^3\ T_3^4\ T_3^5\ T_3^6\ ...\ T_3^{16}\ T_3^{17}$$
$$TPhysical_4 : T_4^1\ T_4^2\ T_4^3\ T_4^4\ T_4^5\ T_4^6\ ...\ T_4^{16}\ T_4^{17}$$
$$TPhysical_5 : T_5^1\ T_5^2\ T_5^3\ T_5^4\ T_5^5\ T_5^6\ ...\ T_5^{16}\ T_5^{17}$$

Figure 9.19: Trace de cinq horloges abstraites $TPhysical_1$, $TPhysical_2$, $TPhysical_3$, $TPhysical_4$ et $TPhysical_5$ simulant respectivement l'activité des processeurs `Proc1`, `Proc2`, `Proc3`, `Proc4` et `Proc5`.

Ordonnancement de tâches représentants des parties d'images sur des processeurs

Nous considérons maintenant le découpage d'image en parties que nous appelons sous-images. Ainsi, dans la Figure 9.20, nous réservons le processeur `Proc1` (resp. `Proc2`) pour le traitement de la première (resp. deuxième) moitié de l'image qui subit l'encodage JPEG. Ce cas d'association est modélisé en Marte dans la Figure 9.12 (exemple 6).

$$TPhysical_1 : T_1^1\ T_2^1\ T_3^1\ T_4^1\ T_5^1\ T_1^2\ T_2^2\ T_3^2\ T_4^2\ T_5^2\ T_5^2\ T_1^3\ T_2^3\ ...\ T_1^8\ T_2^8\ T_3^8\ T_4^8\ T_5^8$$
$$TPhysical_2 : T_1^9\ T_2^9\ T_3^9\ T_4^9\ T_5^9\ T_1^{10}\ T_2^{10}\ T_3^{10}\ T_4^{10}\ T_5^{10}\ ...\ T_1^{17}\ T_2^{17}\ T_3^{17}\ T_4^{17}\ T_5^{17}$$

Figure 9.20: Trace de deux horloges abstraites $TPhysical_1$ et $TPhysical_2$ simulant respectivement l'activité des processeurs `Proc1` et `Proc2` en découpant l'image en deux parties.

De la même manière, la Figure 9.21 représente le cas d'une association des cinq processeurs `Proc1`, `Proc2`, $\texttt{Proc3}_1$, $\texttt{Proc3}_2$ et $\texttt{Proc3}_3$ pour l'exécution de cinq sous-images de l'image qui subit l'encodage JPEG. La modélisation Marte de cette association est illustrée dans la Figure 9.13 (exemple 9).

$$TPhysical_1 : T_1^1\ T_2^1\ T_3^1\ T_4^1\ T_5^1\ T_1^2\ T_2^2\ T_3^2\ T_4^2\ T_5^2\ T_5^2\ T_1^3\ T_2^3\ T_3^3\ T_4^3\ T_5^3\ T_1^4\ T_2^4\ T_3^4\ T_4^4\ T_5^4$$
$$TPhysical_2 : T_1^5\ T_2^5\ T_3^5\ T_4^5\ T_5^5\ T_1^6\ T_2^6\ T_3^6\ T_4^6\ T_5^6\ T_1^7\ T_2^7\ T_3^7\ T_4^7\ T_5^7$$
$$TPhysical_3 : T_1^8\ T_2^8\ T_3^8\ T_4^8\ T_5^8\ T_1^9\ T_2^9\ T_3^9\ T_4^9\ T_5^9\ T_1^{10}\ T_2^{10}\ T_3^{10}\ T_4^{10}\ T_5^{10}$$
$$TPhysical_4 : T_1^{11}\ T_2^{11}\ T_3^{11}\ T_4^{11}\ T_5^{11}\ T_1^{12}\ T_2^{12}\ T_3^{12}\ T_4^{12}\ T_5^{12}\ T_1^{13}\ T_2^{13}\ T_3^{13}\ T_4^{13}\ T_5^{13}$$
$$TPhysical_5 : T_1^{14}\ T_2^{14}\ T_3^{14}\ T_4^{14}\ T_5^{14}\ ...\ T_1^{17}\ T_2^{17}\ T_3^{17}\ T_4^{17}\ T_5^{17}$$

Figure 9.21: Trace de cinq horloges abstraites $TPhysical_1$, $TPhysical_2$, $TPhysical_3$, $TPhysical_4$ et $TPhysical_5$ simulant respectivement l'activité des processeurs `Proc1`, `Proc2`, `Proc3`, `Proc4` et `Proc5`.

9.6.3 Estimation de charges de travail par processeur

Dans cette section, nous estimons la charge de travail pour chaque processeur impliqué dans l'encodage JPEG d'une image de taille (256×256). Cette charge de travail est représentée en terme de cycles d'horloge processeur nécessaire pour l'exécution des fonctionnalités.

Étant donné une trace d'horloges abstraites étendues, le nombre total de valeurs ternaires (occurrence d'une valeur 0, 1 ou -1) des différentes horloges représente le nombre total de cycles d'horloge processeur nécessaire pour l'exécution des fonctionnalités. Par exemple, dans le cas de l'ordonnancement mono-tâche/mono-processeur de l'encodeur JPEG, nous pouvons déduire la charge de travail du processeur `Proc1` en calculant la tailles des différentes tranches associées à l'horloge $TPhysical_5$:

- $taille(T_1^1) = 64 \times c_1$
- $taille(T_1^j) = 64 \times c_1$, où $j \in [2, 16]$
- $taille(T_1^{17}) = 4$
- $taille(T_2^1) = 63 \times c_2 + 1$
- $taille(T_2^j) = 64 \times c_2$, où $j \in [2, 16]$
- $taille(T_2^{17}) = c_2 + 3$
- ...
- $taille(T_5^1) = 60 \times c_5 + 4$
- $taille(T_5^j) = 64 \times c_5$, où $j \in [2, 16]$
- $taille(T_5^{17}) = c_5 \times 4$

Le Tableau 9.3 illustre le nombre de cycles d'horloge processeur pour plusieurs configurations de l'architecture et de l'association. Cette classification prend en compte, d'une part, le nombre de processeurs considérés dans une architecture, et d'autre part, les deux types d'associations processeurs/tâches et sous-images/processeurs.

Dans le Tableau 9.3, nous avons considéré les coûts de communications et d'accès mémoire. Cependant, nous ne prenons pas en compte le surcoût de la fonction d'accès mémoire quand le nombre de processeurs, impliqués dans l'exécution, augmente. Pour cela, nous définissons dans la section suivante une fonction qui estime un surcoût de communication relatif au nombre de processeurs considérés dans la configuration.

Surcoût de la fonction d'accès mémoire

Dans une architecture multiprocesseur à mémoire partagée, une communication entre des processeurs et une mémoire partagée passe à travers un bus. En augmentant le nombre de processeurs, les communications entre les différents processeurs et la mémoire sont surchargées et entraînent des coûts supplémentaires. Quand le nombre de processeurs actifs devient très élevé, il y a un risque de saturation des accès mémoire.

Étant donné un processeur $Proc_i$, nous définissons $T_{comm_i}(x)$ le surcoût de communication et d'accès mémoire pour l'activité de $Proc_i$, contenu dans un ensemble de x

Exemples de configurations	Processeurs	Cycles d'horloge
1	Proc1	32529428
2	Proc1	22188044
	Proc2	10341384
3	Proc1	7714820
	Proc2	7387140
	Proc3	17427468
4	Proc1	7714820
	Proc2	7387140
	Proc3	7086084
	Proc4	10341384
5	Proc1	7714820
	Proc2	7387140
	Proc3	7086084
	Proc4	5637124
	Proc5	4704260
6	Proc1	16208769
	Proc2	16320659
7	Proc1	14175681
	Proc2	12198528
	Proc3	6155219
8	Proc1	8076417
	Proc2	8132352
	Proc3	8132352
	Proc4	8188307
9	Proc1	8076417
	Proc2	6099264
	Proc3	6099264
	Proc4	6099264
	Proc5	6155219

Table 9.3: Cycles d'horloge processeur pour les 9 configurations de l'architecture et de l'association.

processeurs. Il est exprimé en terme de cycles d'horloge du processeur $Proc_i$, selon la fonction suivante :

$$T_{comm}(Proc_i, x) = (x - 1) \times \alpha \times T_{comp}(Proc_i, x) \tag{9.1}$$

où x représente le nombre total de processeurs considérés dans l'exécution, $\alpha = 0.017388$ est une constante que nous avons estimé à partir des simulations de bas niveau, et $T_{comp}(Proc_i, x)$ représente le nombre de cycles d'horloge associé au processeur $Proc_i$ nécessaire pour finir le traitement.

En effet, nous considérons une architecture multiprocesseur contenant au total n processeurs homogènes. Plusieurs configurations d'architecture et d'associations peuvent être considérées. Par exemple, nous pouvons imaginé une exécution de fonctionnalités avec un seul processeur ou avec n processeurs. Nous définissons C_i le nombre de cycles d'horloge processeur totale pour l'exécution d'une fonctionnalité, contenant i processeurs[6]. Nous définissons $C_{moyenne}$ comme étant la moyenne des cycles d'horloges totales de toutes les configurations possibles (Formule 9.2).

$$C_{moyenne} = \frac{\sum_{i=1}^{n} C_i}{n} \tag{9.2}$$

Nous définissons $\delta_{C_i,C_{i+1}}$ le surcoût en terme de cycles d'horloge entre une exécution de fonctionnalité avec i et $i + 1$ processeurs (Formule 9.3).

$$\delta_{C_i,C_{i+1}} = \frac{C_i + C_{i+1}}{2} \tag{9.3}$$

La variable $\delta_{moyenne}$ représente la moyenne des valeurs δ :

$$\delta_{moyenne} = \frac{\sum_{i=1}^{n-1} \delta_{C_i,C_{i+1}}}{n - 1} \tag{9.4}$$

Enfin, nous définissons α comme étant le rapport entre la moyenne des cycles d'horloge processeur représentant le surcoût de communication et le nombre moyen de cycles d'horloge du processeur.

$$\alpha = \frac{\delta_{moyenne}}{C_{moyenne}} \tag{9.5}$$

Ainsi, le nombre total de cycles d'horloge processeur $Cycles_i(x)$, quand x processeurs sont considérés est :

$$Cycles(Proc_i, x) = T_{comp}(Proc_i, x) + T_{comm}(Proc_i, x) = T_{comp}(Proc_i, x)(1 + (x - 1) \times \alpha) \tag{9.6}$$

En appliquant la fonction $Cycles(x)$ sur les résultats illustrés dans le Tableau 9.3, nous obtenons maintenant de nouvelles valeurs de cycles d'horloge processeur pour les différentes configurations. Ces valeurs sont illustrées dans le Tableau 9.4. Il faut noter que cette fonction d'estimation du surcoût d'accès mémoire n'est valable que pour un nombre limité de processeurs. Elle ne serai plus applicable quand la mémoire partagée devient saturée.

[6]Les valeurs C_i sont obtenues à partir des mesures de bas niveau (TLM ou RTL).

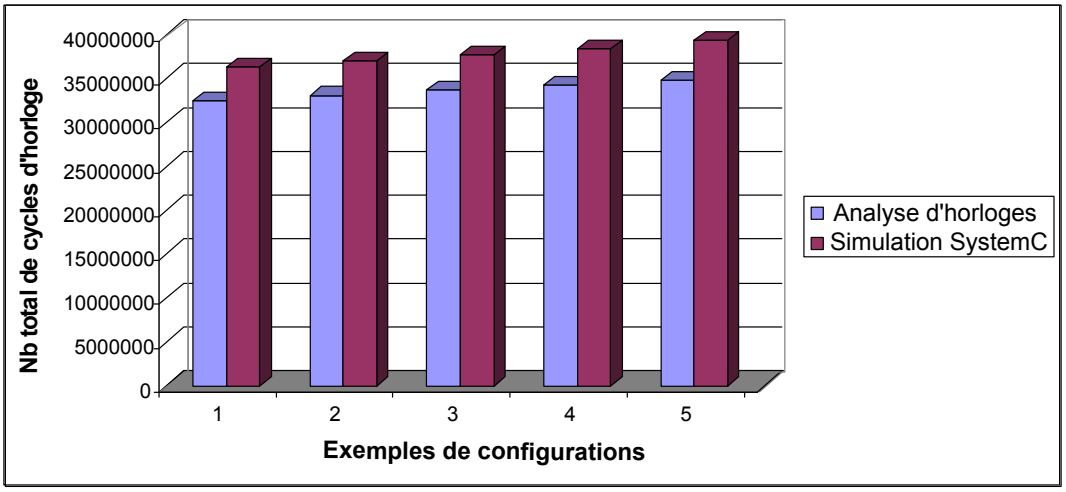


Figure 9.22: Comparaison des résultats de nombres de cycles d'horloge obtenus à partir d'une analyse d'horloges abstraites et de simulations en SystemC selon différentes configurations de l'architecture.

Exemple. Prenons le cas de l'exemple 2. Deux processeurs `Proc1` et `Proc2` ont respectivement $T_{comp_1} = 22188044$ et $T_{comp_2} = 10341384$ cycles d'horloge. Ces cycles d'horloge représentent les charges de travail des deux processeurs, sans prendre en compte les coûts de communications. Pour cela nous appliquons les fonctions $C_1(x)$ et $C_2(x)$ sur les cycles d'horloge T_{comp_1} et T_{comp_2}, où $x = 2$:

- $T_{comm}(Proc_1, 2) = (1) \times 0.017388 \times 22188044 = 385806$
- $T_{comm}(Proc_2, 2) = (1) \times 0.017388 \times 10341384 = 179816$

Le nombre de cycles d'horloge associé aux processeurs `Proc1` et `Proc2` devient :

- $Cycles(Proc_1, 2) = 22188044 + 385806 = 22573850$
- $Cycles(Proc_2, 2) = 10341384 + 179816 = 10521200$

La Figure 9.22 montre une comparaison de nombre de cycles d'horloge processeur obtenus en appliquant l'analyse d'horloges abstraites (barres de couleur bleu) et des simulations de l'encodeur JPEG en SystemC au niveau TLM (barres de couleur mauve). La comparaison est faite pour des architectures allant de 1 jusqu'à 5 processeurs. Nous remarquons que les deux types de barres ont tendances à croître quand le nombre de processeurs augmente. Cela est dû au surcoût de communication qui augmente à chaque fois que le nombre de processeur augmente.

9.6.4 Analyse de propriétés non fonctionnelles

9.6.4.1 Estimation du temps d'exécution des différentes configurations

Dans cette section, nous estimons le temps d'exécution d'un encodage JPEG d'une image de taille (256 × 256) et selon plusieurs configurations de l'architecture et de l'association. Nous considérons une fréquence commune, de 300 MHz, pour l'ensemble des processeurs.

Exemples de configurations	Processeur	Cycles d'horloge
1	Proc1	32529428
2	Proc1	22573850
	Proc2	10521200
3	Proc1	7983111
	Proc2	7644035
	Proc3	18033526
4	Proc1	8117256
	Proc2	7772483
	Proc3	7455722
	Proc4	10880832
5	Proc1	8251401
	Proc2	7900930
	Proc3	7578935
	Proc4	6029197
	Proc5	5031451
6	Proc1	16490607
	Proc2	1660443
7	Proc1	14668654
	Proc2	12622744
	Proc3	6369273
8	Proc1	8497715
	Proc2	8556568
	Proc3	8556568
	Proc4	8615442
9	Proc1	8638148
	Proc2	6523480
	Proc3	6523480
	Proc4	6523480
	Proc5	6583327

Table 9.4: Cycles d'horloge processeur pour les 9 configurations de l'architecture et de l'association en considérant les surcoûts d'accès à la mémoire par rapport au nombre de processeurs.

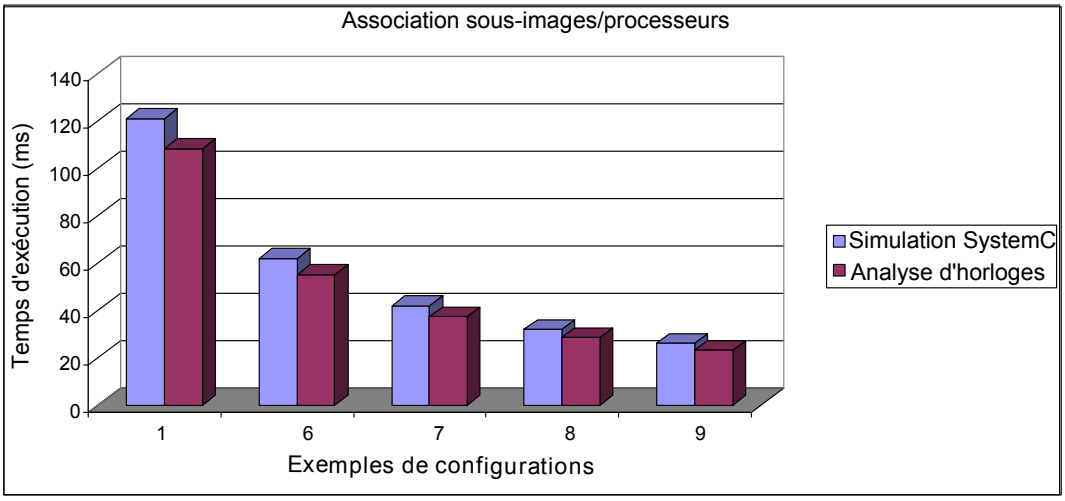


Figure 9.23: Comparaison de temps d'exécution pour différentes configurations de l'architecture (nombre de processeurs) et de l'association de sous-images sur différents processeurs, obtenus durant l'analyse d'horloges abstraites (barres de couleur mauve) et des simulations en SystemC au niveau TLM (barres de couleur bleu).

Associations de processeurs à des sous-images

Dans la Figure 9.23, les temps d'exécutions de cinq configurations de l'architecture multiprocesseur, contenant de un jusqu'à cinq processeurs, sont définis. L'image est découpée en des parties ou sous-images. Pour chaque configuration, chaque sous-image est exécutée par un processeur de fréquence 300 MHz. Les barres, de couleur bleu, représentent les durées d'exécutions d'une image selon les différentes configurations de l'architecture. Ces durées sont obtenues à partir de simulations en SystemC. Les barres, de couleur mauve, représentent les durées d'exécutions de la même image en appliquant une analyse de systèmes à base d'horloges abstraites.

Observation : nous remarquons que les types de barres ont presque les mêmes résultats. Cela revient à la régularité de traitement quand des processeurs sont alloués à différentes sous-images. Pour ce type d'associations, nous sommes en mesures de prévoir et d'estimer les durées d'exécution en un coût presque négligeable par rapport à d'autre simulation comme celles faites en SystemC. Les résultats obtenus en appliquant la méthodologie d'analyse d'horloges abstraites sont aussi précis que les résultats obtenus en simulant les mêmes configurations au niveau transactionnel (TLM). Nous concluons de ce graphe que les configurations de deux et de trois processeurs sont celles qu'il faut choisir. En effet, en passant d'une configuration contenant un processeur à une configuration ayant deux processeurs, le temps d'exécution total de l'application diminue de presque 50%. De même, en passant de deux processeurs à trois processeurs, le temps d'exécution diminue de 33%. Ces deux configurations améliorent considérablement les performances du système en intégrant un nombre raisonnable de processeurs. Au delà de trois processeurs, les gains en performance par rapport au nombre additionnel de processeurs n'est plus apprécié.

Association de processeurs aux tâches

Dans la Figure 9.24, cinq configurations de l'architecture multiprocesseur sont définies. Cependant, nous considérons cette fois-ci des associations de types tâches/processeurs. Par exemple, dans le cas d'une architecture à deux processeurs, nous allouons les tâches `rgb`, `dct` et `qu` au premier processeur et les tâches `qu` et `re` au deuxième processeur. Les barres, de couleur bleu, représentent les durées d'exécutions obtenues à partir de simulations en SystemC. Les barres de couleur mauve sont obtenues à partir d'une analyse du système basée sur les horloges abstraites.

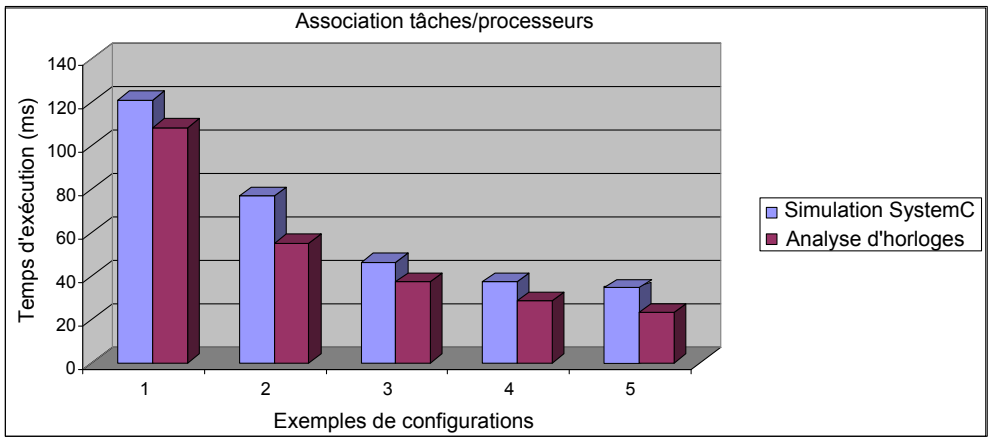


Figure 9.24: Comparaison de temps d'exécution entre une analyse d'horloges abstraites (barres de couleur mauve) et des simulations en SystemC (barres de couleur bleu) pour différentes associations de tâches fonctionnelles exécutées par différents processeurs.

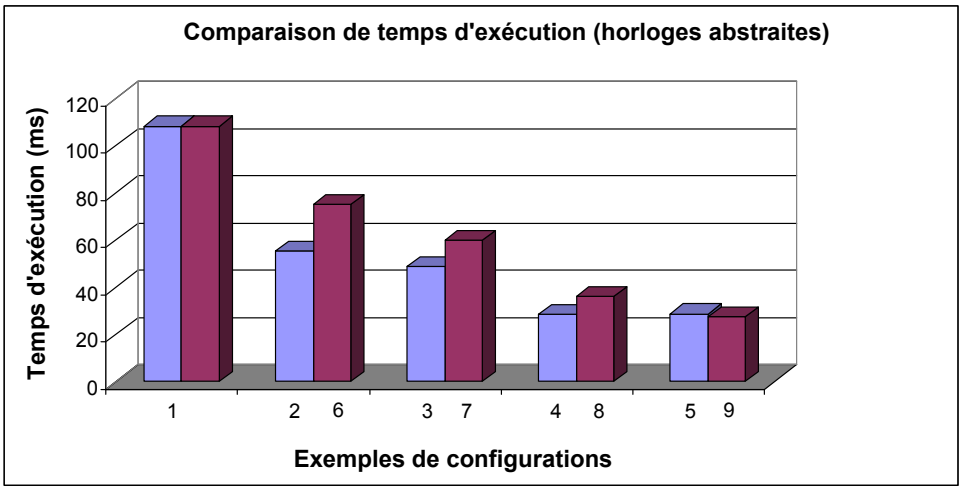
Observation : nous remarquons de même que les deux types de barres (bleu et mauve) ont les mêmes tendances pour les configurations de l'architecture ayant un, trois et quatre processeurs. Quant à la configuration ayant deux processeurs, nous remarquons une grande différence entre les deux barres. Cela est dû probablement à la saturation de la mémoire partagée entre les différents processeurs (observation à partir des résultats de simulation en SystemC). Cette irrégularité est indétectable en appliquant l'analyse d'horloges abstraites.

Graphe comparatif

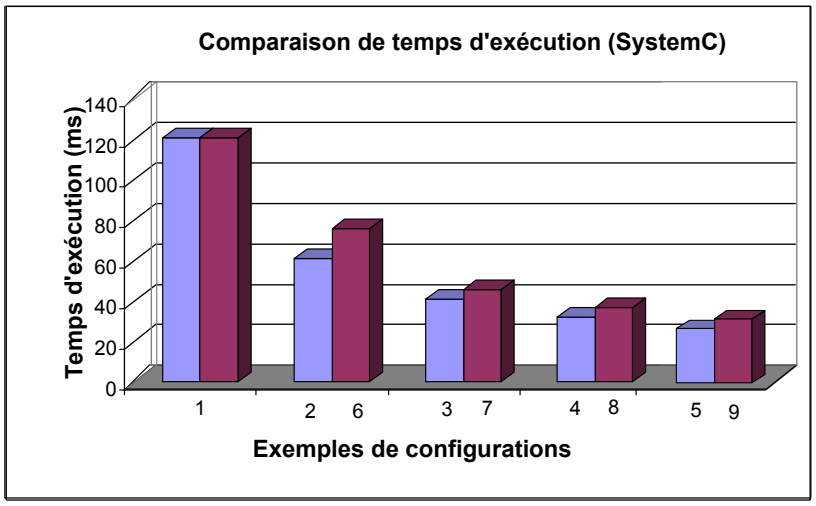
Nous présentons dans la Figure 9.25 deux graphes comparatifs de temps d'exécutions de la compression JPEG d'une image de taille (256×256) :

- dans la Figure 9.25 a), nous comparons les temps d'exécutions des deux types d'allocations tâches/processeurs (barres de couleur mauve) et sous-images/processeurs (barres de couleur bleu). Ces résultats sont obtenus en appliquant la méthodologie d'analyse d'horloges abstraites que nous avons proposée dans cette thèse ;
- dans la Figure 9.25 b), nous simulons différents scénarios d'exécution, au niveau transactionnel (TLM) en SystemC.

En analysant chacun des deux graphes, nous remarquons que les associations de type sous-images/processeurs, en terme de temps d'exécution, sont beaucoup plus avantageuses que les associations de type tâches/processeurs. Nous pouvons aussi



(a)



(b)

Association sous-images/processeurs
Association tâches/processeurs

Configurations :
1 : exécution mono-processeur
2,6 : exécution deux-processeurs
3,7 : exécution trois-processeurs
4,8 : exécution quatre-processeurs
5,9 : exécution cinq-processeurs

Figure 9.25: Légende: (a) Comparaison de temps d'exécution entre les deux types d'associations tâches/processeurs (exemple 1, 2, 3, 4, 5) et sous-images/processeurs (exemple 1, 6, 7, 8, 9), selon l'analyse d'horloges abstraites et (b), comparaison de temps d'exécution entre les deux types d'associations tâches/processeurs (exemple 1, 2, 3, 4, 5) et sous-images/processeurs (exemple 1, 6, 7, 8, 9) en SystemC.

déduire que les configurations de l'architecture contenant deux et trois processeurs sont à garder. Les configurations contenant un, quatre et cinq processeurs sont à rejeter.

9.6.4.2 Détermination de fréquences minimales

Dans l'étude de cas que nous présentons, nous pouvons imaginer une option appelée *Mode Débit Image*. Cette option est représentée par une variable portant les valeurs dix, quinze et vingt. Ces valeurs fixent le débit d'image par seconde pour un encodage JPEG d'images de taille (256×256). Nous considérons, dans ce qui suit, le cas de séquences finies d'images encodées selon un débit constant de 15 images par seconde. Cela signifie que la durée maximale pour encoder une image est de $1/15 \simeq 0.066$ secondes.

La fréquence minimale d'un processeur est celle qui assure les propriétés suivantes :

- le traitement de données et la production des résultats souhaités avant les temps d'échéances imposés sur l'application ;
- préservation de contraintes de dépendances de tâches et de données ;
- réduction autant que possible l'intervalle de temps entre la fin de traitements et les temps d'échéance.

Étant donné un système où les contraintes de dépendances de tâches sont validées par construction (par exemple une architecture mono-processeur ou une association de sous-images à des processeurs), il suffit d'appliquer la Formule 7.9, définis dans le chapitre 7, afin de trouver la fréquence minimale d'un processeur $Proc_i$:

$$f_{min}(Proc_i) = \frac{ct_i}{dl_i}$$

où ct_i représente le nombre de cycle d'horloge total associé à $proc_i$ et dl_i est le temps d'échéance imposé sur la terminaison de l'exécution de fonctionnalité.

Dans le cas d'ordonnancement de sous-images aux différents processeurs impliqués dans l'exécution, les contraintes de dépendances de tâches sont respectées par construction. En effet, les cinq horloges étendues clk'_1, clk'_2, clk'_3, clk'_4 et clk'_5 sont découpées en des tranches ayant la même taille. Ces tranches sont ensuite allouées par ordre sur les différents processeurs en commençant toujours l'exécution par les horloges productrices de données. Le Tableau 9.5 contient les fréquences minimales associées aux différents cas des architecture et pour une association de sous-images sur des processeurs.

Calcul de ratios de fréquences

Les dépendances de données, explicitement spécifiées dans une trace d'horloges abstraites fonctionnelles, peuvent être violées après l'expansion des horloges abstraites fonctionnelles.

En effet, dans une trace d'horloges abstraites fonctionnelles, des contraintes de précédences et de périodicités sont imposées. Par exemple, dans la trace d'horloges abstraites associée au fonctionnalité de l'encodeur JPEG, la $n^{\text{ième}}$ activation de la tâche *rgb*, représentée par la $n^{\text{ième}}$ occurrence d'une valeur 1 dans l'horloge clk_1, doit être précédée par la $n^{\text{ième}}$ activation de la tâche *dct*. Ces contraintes peuvent être vues comme un comportement périodique comme suit :

$$clk_i(n) = clk_{i-1}(n) + 1, \text{ où } i \in [1, 5]$$

Processeur	Cycles d'horloge	Fréquence minimale (MHz)
Proc1	32529428	493
Proc1	16490607	250
Proc2	16604443	252
Proc1	14668654	223
Proc2	12622744	192
Proc3	6369273	97
Proc1	8497715	129
Proc2	8556568	130
Proc3	8556568	130
Proc4	8615442	131
Proc1	8638148	131
Proc2	6523480	99
Proc3	6523480	99
Proc4	6523480	99
Proc5	6583327	100

Table 9.5: Fréquences minimales des processeurs pour cinq cas de configuration d'architecture, en considérant une association de sous-images par processeurs.

Afin de vérifier les contraintes de dépendances de tâches fonctionnelles, nous régularisons les fréquences des différents processeurs de telle sorte que ces contraintes soient toujours vérifiées. Nous calculons d'abord les grandes périodes entre les couples d'horloges abstraites fonctionnelles ayant des canaux de communications. Pour cela, nous appliquons la formule de la grande période (définition 17 du chapitre 7) sur les couples d'horloges (clk_1, clk_2), (clk_2, clk_3), (clk_3, clk_4) et (clk_4, clk_5). Le nombre de cycles d'horloge entre deux occurrences de la valeur 1 dans clk'_i est $inb_i = c_i$ et sa grande période est $GP_i = inb_i + 1$. Nous obtenons ainsi les valeurs suivantes :

- $Ratio(f_1, f_2) = \frac{GP_1}{GP_2} = \frac{c_1+1}{c_2+1} = 1.044$;
- $Ratio(f_2, f_3) = \frac{GP_2}{GP_3} = \frac{c_2+1}{c_3+1} = 1.042$;
- $Ratio(f_3, f_4) = \frac{GP_3}{GP_4} = \frac{c_3+1}{c_4+1} = 1.257$;
- $Ratio(f_4, f_5) = \frac{GP_4}{GP_5} = \frac{c_4+1}{c_5+1} = 1.198$;

9.6.4.3 Estimation de la puissance dissipée

Afin de pouvoir calculer la consommation d'énergie d'un système, nous devons d'abord calculer la puissance dissipée pour chaque processeur impliqué dans l'exécution. Afin d'y arriver, nous devons savoir les couples de valeurs (fréquence, tension) pour chaque type de processeur impliqué dans l'exécution. Cette information peut varier d'un type de processeur à un autre. Le Tableaux 9.6 représentent les valeurs de tensions pour des intervalles de fréquences, d'un processeur du type `PowerPC 405 CPU Core`. Ces informations sont extraites directement de la spécification du processeur [60].

Pour calculer les valeurs de puissances dynamiques dissipées, nous appliquons la Formule 7.12[7]. En fixant la valeur de capacité $C = 0.0127$, nous avons toutes les informations nécessaires pour estimer la puissance dissipée de chaque processeur impliqué

[7] $P_{dynamique} = C \times f \times V^2$

Intervalle de fréquence (MHz)	tension (Volt)
1-140	0.7
141-288	0.9
289-433	1.1
434-658	1.2

Table 9.6: Estimation des couples de valeurs (fréquence, tension) pour un processeur du type `PowerPC 405 CPU Core`.

dans une exécution (capacité, fréquence et tension). Nous calculons ainsi les valeurs de puissances dissipées pour différentes configurations de l'architecture. Le Tableau 9.7 résume les différentes valeurs de puissances obtenues.

Afin d'analyser les résultats, nous montrons dans la Figure 9.26 les valeurs de puissances totales dissipées pour différentes configurations de l'architecture et de l'association. Ces valeurs sont obtenues en appliquant la formule de la puissance sur les résultats obtenus dans l'analyse d'horloges abstraites. Nous concluons de ce graphe que les configurations de deux, trois et quatre processeurs sont pertinentes de points de vue dissipation de puissances dynamiques. En passant d'une première configuration, contenant un seul processeur avec une fréquence de 493 MHz, à une deuxième configuration contenant deux processeurs ayant respectivement les fréquences 250 et 252 MHz, nous diminuons la puissance dissipée totale du système d'environ 43%. De la même manière, en passant d'une configuration de trois processeurs à une configuration de quatre processeurs, nous diminuons la puissance totale dissipée d'environ 33%. Cependant, ces résultats sont purement théoriques et n'ont pas été validés par des simulations de bas niveau.

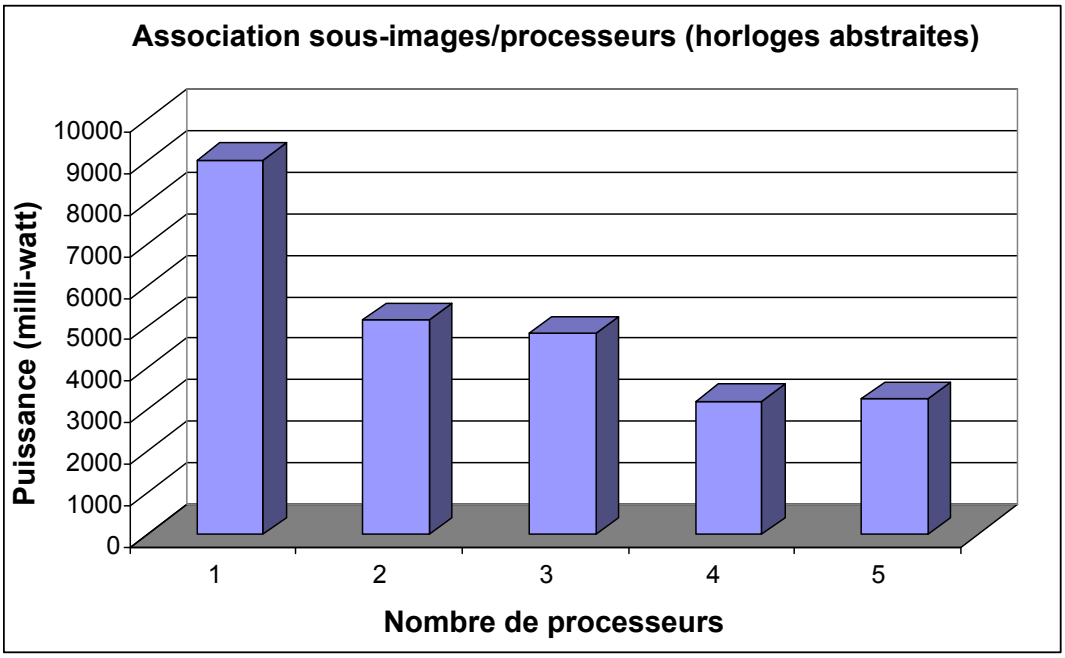


Figure 9.26: Comparaison de puissances dissipées pour différentes configurations de l'architecture et de l'association.

Processeur	Fréquence minimale (MHz)	Puissance (MilliWatt)
Proc1	493	9015
Proc1	250	2570
Proc2	252	2590
Proc1	223	2280
Proc2	192	1960
Proc3	97	600
Proc1	129	800
Proc2	130	800
Proc3	130	800
Proc4	131	810
Proc1	131	810
Proc2	99	610
Proc3	99	610
Proc4	99	610
Proc5	100	620
Proc1	100	620
Proc2	101	620
Proc3	101	620
Proc4	101	620
Proc5	101	620
Proc6	35	210

Table 9.7: Estimation des puissances dynamiques de processeurs selon différents types d'associations sous-images/processeurs.

9.7 Conclusion

Dans ce chapitre, nous avons proposé une étude de cas d'un encodeur JPEG, implanté sur du MPSoC. Cela fut l'occasion de montrer l'usage des propositions faites dans les chapitres 5, 6 et 7. Nous avons d'abord modélisé une application en Marte, consistant d'un encodage JPEG d'images de tailles (256×256) pixels. Ensuite, nous avons proposé une architecture multiprocesseur de type PRAM comme support d'exécution. La modélisation de l'application JPEG est complètement indépendante de la modélisation de l'architecture. C'est durant l'association de fonctionnalités sur des ressources physiques que ces deux vues s'entrelacent. Nous avons donc proposé plusieurs configurations d'associations possibles pour enrichir l'étude de cas. L'objectif visé est de choisir les meilleures configurations de l'architecture et de l'association selon des critères de performances.

Des contraintes temporelles ont été considérées au niveau fonctionnel. Elles étaient extraites après avoir considéré un modèle d'exécution pipeline pour l'application. Nous avons ensuite décrit les fonctionnalités JPEG, modélisés en Marte, par le biais d'horloges fonctionnelles. Les horloges fonctionnelles, considérées comme étant synchrones, sont ensuite projetées sur des horloges physiques, potentiellement asynchrone. Le résultat de cette projection est une trace d'horloges d'exécution, simulant l'activité des processeurs durant l'exécution de fonctionnalités. Cette projection s'est basée sur un algorithme d'ordonnancement, de type statique non-préemptif.

Nous avons ensuite appliqué une méthodologie d'analyse, à base d'horloges abstraites, afin d'évaluer rapidement des propriétés fonctionnelles et non fonctionnelles. À partir de cette analyse, nous avons réussi à réduire l'espace de solutions possibles de configurations des architectures et des associations en se basant sur plusieurs critères, tels que le temps d'exécution et la puissance dissipée. Nous avons comparé les résultats obtenus avec des simulations en SystemC au niveau transactionnel. Nous avons trouvé que les constats sur les configurations idéales des architectures et des associations, obtenus à partir de l'analyse d'horloges abstraites, sont aussi valables pour les simulations SystemC.

Ce fut également l'opportunité de démontrer le fonctionnement de l'abstraction d'un système, co-modélisé en UML/ Marte, en un formalisme d'horloges abstraites. Cette abstraction a permis d'analyser différents paramètres de l'architecture et de l'association. Nous avons vu que, grâce à cette transformation vers un langage temporel, il est particulièrement facile et rapide d'explorer l'espace de solutions architecturales à un haut niveau d'abstraction.

Conclusion et perspectives

La contribution générale

Les travaux que nous venons de présenter s'inscrivent dans le cadre du développement de systèmes sur puce multiprocesseur MPSoC dédiés à l'exécution d'applications hautes performances, comme celles du traitement de signal intensif. L'analyse et la vérification de systèmes, co-modélisés à un haut niveau d'abstraction, représentent la brique de base de ce travail. L'objectif final est de réduire l'espace de solutions possibles de configurations d'architectures et d'associations. Seuls les solutions pertinentes sont retenues afin de les analyser ultérieurement dans des niveaux plus complexes et détaillés.

Nous avons d'abord exposé les difficultés actuelles dans la conception de MPSoC, surtout quand des applications hautes performances sont considérées, ainsi que des architectures massivement parallèles. Nous avons présenté un état de l'art qui couvre une grande partie de travaux récents dans ce domaine. Nous avons classifié les différents travaux existants dans l'état de l'art selon certains critères que nous avons jugés importants.

Il est important de souligner, que tout au long de cette thèse, nous avons prêté une grande attention aux méthodologies de conception basées sur les modèles afin de résoudre le problème de l'augmentation de la complexité de conception. À travers l'environnement Gaspard2, nous avons présenté une partie du profil Marte. Ce profil permet de spécifier un graphe de tâches fonctionnelles, avec une grande expressivité de parallélismes de tâches et de données. Cela est possible en se servant du paquetage RSM de Marte. Nous avons aussi montré comment modéliser des architectures massivement parallèles.

De nos travaux de thèse, il résulte une analyse, à un haut niveau d'abstraction, de systèmes co-modélisés en Marte. À partir d'une analyse d'horloges abstraites, nous arrivons à reproduire l'activité des différents processeurs concernés dans l'exécution de fonctionnalités. Cette activité est exprimée en terme de cycles d'horloge processeur. D'une part, cela nous permet de vérifier l'ordre d'exécution de tâches en analysant des contraintes de périodicités et de précédences. D'autre part, nous proposons une analyse de propriétés non fonctionnelles telles que le respect des temps d'échéances, l'estimation du temps d'exécution et de la consommation d'énergie. Les résultats obtenus à partir de cette analyse permettront de prendre des décisions sur les différentes configurations possibles de systèmes à retenir.

Les configurations retenues seront ultérieurement analyser par des simulations et de prototypages de bas niveau. Nous avons validé notre proposition ainsi que les résultats par une étude de cas d'un encodeur JPEG. Nous avons comparé les résultats obtenus à partir de l'analyse d'horloges abstraites avec des simulations faites au niveau transactionnel en SystemC. Nous avons eu des résultats très encourageants, vu la facilité et la rapidité de l'analyse d'horloges abstraites par rapport à d'autres types d'analyse et de simulation.

Perspectives

Les travaux actuels et les perspectives peuvent être classifiés en quatre parties :

1. l'automatisation complète de la méthodologie d'analyse et de vérification d'horloges abstraites. Cela aboutira à un outil intelligent permettant de restreindre le nombre de solutions possibles à celles considérées idéales (selon un ensemble de critères) ;
2. utilisation des algorithmes d'ordonnancements usuels afin d'obtenir des horloges abstraites d'exécutions ;
3. considération d'architectures multiprocesseur à mémoire distribuée ;
4. l'enrichissement du modèle d'analyse en ajoutant plus de détails d'implémentations. Cet enrichissement peut être fais à travers une librairie de modèles propres à l'analyse multi-critères de systèmes multiprocesseur.

Automatisation de l'exploration de l'espace de conception

Nous avons proposé un prototype d'analyse de systèmes manipulant des horloges abstraites. Ce prototype, à base de fonction OCaml, prend en entrée des couples d'horloges fonctionnelles ou d'exécution. Cependant, cet outil reste de nature primitif et nécessite des améliorations. Afin d'avoir une automatisation complète de la méthodologie d'analyse d'horloges abstraites, ce prototype doit être branché au modèle UML afin d'analyser automatiquement le système complet. Cette étape n'est pas encore complète et nécessite encore d'efforts d'ingénierie dirigée par des modèles. Aussi, le branchement de cet outil dans un environnement d'exploration d'espace de conception est primordial. Nous imaginons par exemple un environnement d'exploration d'architectures et d'associations à base d'algorithmes génétiques. En branchant l'outil à un tel environnement, nous arriverons en peu de temps à trouver les meilleures configurations d'architectures et d'associations.

Utilisation d'algorithmes d'ordonnancements usuels

Afin de projeter des horloges fonctionnelles sur des horloges physiques, nous avons défini un algorithme d'ordonnancement de type statique non préemptif. Cet algorithme nous a permis de redistribuer les instants logiques d'une horloge fonctionnelle selon les rythmes des différents processeurs. Cependant, nous aurions pu trouver un moyen de se servir des algorithmes d'ordonnancement usuel, tels que EDF et RM. Cela nous permet de tirer profit de nombreux travaux et extensions de ces algorithmes.

Considération des architectures multiprocesseur à mémoires distribuées

Durant la modélisation d'architecture, nous n'avons considéré que des architectures multiprocesseur à mémoires partagées. Cela nous a facilité l'analyse comme les accès mémoires sont considérés équitables. Cependant, les architectures multiprocesseur à mémoire partagée ont un inconvénient majeur. Ce dernier est représenté par le nombre limité de processeurs dans une architecture. En effet, une fois le nombre de processeurs dépassant une certaine limite, les accès mémoires deviennent parfois presque impossibles. Cela est dû à la congestion du bus partagé par les différents processeurs. Pour cela, nous devons prendre en considération des systèmes à mémoires distribuées afin de surmonter le nombre limité de processeurs dans une architecture à mémoire partagée.

Enrichissement du modèle d'analyse

Le modèle d'analyse nécessite plusieurs informations cruciales afin de simuler l'exécution de processeurs durant l'exécution de fonctionnalités. Notamment, nous avons besoin de savoir, pour chaque composant élémentaire, le nombre de cycle d'horloges processeurs nécessaires pour son exécution. Cette information dépend fortement du type du processeur considéré ainsi que de la quantité de données manipulées. Pour cela, nous devons effectué des mesures sur des cartes réelles ou faire des simulations à plusieurs niveaux d'abstraction d'un grand ensemble de tâches fonctionnelles en général et particulièrement celles connues dans le domaine de traitements d'images (par exemple, FFT, filtre horizontal, filtre vertical, réorganisation de données, etc.).

Nous envisageons aussi une amélioration du modèle de coûts de communications de processeurs avec la mémoire partagée. Ce coût augmente au fur et à mesure que le nombre de processeur augmente. Le modèle que nous avons proposé est valable pour un nombre limité de processeurs.

Bibliography

[1] Leakage current: Moore's law meets static power. *Computer*, 36(12):68–75, December 2003. 36

[2] A. Abdallah. Modélisation UML/MARTE et Analyse Temporelle pour une Exploration Architecturale des Systemes-sur-Puce. September 2009. 75

[3] A. Abdallah, A. Gamatié, and J.-L. Dekeyser. Modélisation uml/marte de soc et analyse temporelle basée sur l'approche synchrone. In *SympA'13: SYMPosium en Architecture de machines*, Toulouse, France, September 2009. 75, 82

[4] A. Abdallah, A. Gamatié, and J.-L. Dekeyser. MARTE-based Design of a Multimedia Application and Formal Analysis. In *FDL'08: Forum on specification and design languages*, Stuttgart, Germany, September 2008. 75, 87

[5] A. Abdallah, A. Gamatie, and J.-L. Dekeyser. Model-driven design of embedded multimedia applications on socs. In *DSD '09: Proceedings of the 2009 12th Euromicro Conference on Digital System Design, Architectures, Methods and Tools*, pages 207–210, Patras, Greece, 2009. IEEE Computer Society. 75, 82

[6] A. Abdallah, A. Gamatié, and J.-L. Dekeyser. Correct and Energy-Efficient Design of SoCs: the H.264 Encoder Case Study. In *SoC'2010: International Symposium on System-on-Chip*, Tampere Finlande, 2010. 75, 82

[7] O. Adeluyi, E. ok Kim, J. Lee, and J.-G. Lee. The use of fair y-sim for optimizing mapping set selection in hardware/software co-design. In *ISOCC '08: SoC Design Conference*, volume 2, pages 174–178, Busan, Corée du Sud, 2008. 19

[8] R. M. Akli. Optimisation multicritère pour le placement d'applications intensives sur système-sur-puce (soc), 2010. Master thesis, Supervised by Professor Pierre Boulet (INRIA/LIFL) and Permanent Researcher Laetitia Jourdan (INRIA/USTL). 69

[9] C. André, F. Mallet, and R. De Simone. Time Modeling in MARTE. In *FDL'07: ECSI Forum on specification & Design Languages*, pages 268–273, Barcelona Spain, 2007. ECSI. Available at : http://www.ecsi-association.org/ecsi/main.asp?l1=library&fn=def&id=268. 58, 78

[10] F. Balarin, Y. Watanabe, H. Hsieh, L. Lavagno, C. Passerone, and A. Sangiovanni-Vincentelli. Metropolis: An integrated electronic system design environment. *IEEE Computer*, 36:45–52, 2003. http://www.embedded.eecs.berkeley.edu/metropolis. 28

[11] D. Bautista, J. Sahuquillo, H. Hassan, S. Petit, and J. Duato. Dynamic task set partitioning based on balancing resource requirements and utilization to reduce power consumption. In *SAC '10: Proceedings of the 2010 ACM Symposium on Applied Computing*, pages 521–526, Sierre, Switzerland, 2010. ACM. 51

[12] R. Ben Atitallah. *Modèle et Simulation des système sur purce multiprocesseurs - Estimation des performances et de la consommation d'énergie*. PhD thesis, USTL, 2008. 61, 68

[13] L. Benini, A. Bogliolo, and G. D. Micheli. System-level dynamic power management. *IEEE Alessandro Volta Memorial Workshop on Low-Power Design*, page 23, 1999. 43

[14] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. de Simone. The synchronous languages twelve years later. *Proc. of IEEE*, 91(1):64–83, January 2003. 78

[15] A. Benveniste, P. Le Guernic, and C. Jacquemot. Programming with events and relations: the signal language and its semantics. *Science of Computer Programming*, 16(2):103–149, November 1991. 59

[16] L. Besnard, T. Gautier, and P. L. Guernic. Signal reference manual, 2007. www.irisa.fr/espresso/Polychrony. 114

[17] S. L. Beux. *Un flot de conception pour applications de traitement du signal systématique implémentées sur FPGA à base d'Ingénierie Dirigée par les Modèles*. PhD thesis, Lille, France, 2007. 62, 69

[18] J. Bézivin. On the unification power of models. In *SoSym'05: Software and System Modeling*, volume 4, pages 171–188, 2005. 67

[19] N. Bombieri, F. Fummi, G. Pravadelli, and J. Marques-Silva. Towards Equivalence Checking Between TLM and RTL Models. In *MEMOCODE '07: Proceedings of the 5th IEEE/ACM International Conference on Formal Methods and Models for Codesign*, pages 113–122, Washington, DC, USA, 2007. IEEE Computer Society. 18

[20] J. Boucaron, R. de Simone, and J.-V. Millo. Formal methods for scheduling of latency-insensitive designs. *EURASIP Journal on Embedded Systems*, 2007:8, 2007. 45, 46

[21] S. BOUKHECHEM. *Contribution à la mise en place d'une plateforme open-source MPSoC sous SystemC pour la Co-simulation d'architectures hétérogènes*. PhD thesis, Université de BOURGOGNE, 2008. 5, 17

[22] P. Boulet. Formal Semantics of Array-OL, a Domain Specific Language for Intensive Multidimensional Signal Processing. Research Report RR-6467, INRIA, 2008. Available at http://hal.inria.fr/inria-00261178/PDF/RR-6467.pdf. 27, 58

[23] D. M. Buede. *The Engineering Design of Systems: Models and Methods*. Wiley Publishing, 2nd edition, 2009. 22

[24] F. Campi, R. König, M. Dreschmann, M. Neukirchner, D. Picard, M. Jüttner, E. Schüler, A. Deledda, D. Rossi, A. Pasini, M. Hübner, J. Becker, and R. Guerrieri. RTL-to-layout implementation of an embedded coarse grained architecture for dynamically reconfigurable computing in systems-on-chip. In *SOC'09: Proceedings of the 11th international conference on System-on-chip*, pages 110–113, Tampere, Finland, 2009. IEEE Press. 4

[25] J. Ceng, W. Sheng, J. Castrillon, A. Stulova, R. Leupers, G. Ascheid, and H. Meyr. A high-level virtual platform for early MPSoC software development. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 11–20, Grenoble, France, 2009. ACM. 19

[26] R. Chakraborty and D. R. Chowdhury. A hierarchical approach towards system level static timing verification of SoCs. In *ICCD'09: Proceedings of the 2009 IEEE international conference on Computer design*, pages 201–206, Lake Tahoe, California, USA, 2009. IEEE Press. 22

[27] A. P. Chandrakasan, S. Sheng, and R. W. Brodersen. Low Power CMOS Digital Design. *IEEE Journal of Solid State Circuits*, 27:473–484, 1995. 36

[28] E. Cheung, H. Hsieh, and F. Balarin. Fast and accurate performance simulation of embedded software for MPSoC. In *ASP-DAC '09: Proceedings of the 2009 Asia and South Pacific Design Automation Conference*, pages 552–557, Yokohama, Japan, 2009. IEEE Press. 19

[29] A. Cohen, M. Duranton, C. Eisenbeis, C. Pagetti, F. Plateau, and M. Pouzet. N-synchronous Kahn networks. In *PoPL'06: ACM Symp. on Principles of Programming Languages*, Charleston, South Carolina, USA, January 2006. 6, 42, 45, 46, 82, 109, 134

[30] A. Cohen, L. Mandel, F. Plateau, and M. Pouzet. Abstraction of Clocks in Synchronous Data-Flow Systems. In *APLAS '08: Proceedings of the 6th Asian Symposium on Programming Languages and Systems*, pages 237–254, Berlin, Heidelberg, 2008. Springer-Verlag. 43

[31] D. Cordes, P. Marwedel, and A. Mallik. Automatic parallelization of embedded software using hierarchical task graphs and integer linear programming. In *CODES/ISSS '10: Proceedings of the eighth IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 267–276, Scottsdale, Arizona, USA, 2010. ACM. Available at: http://doi.acm.org/10.1145/1878961.1879009. 52

[32] R. Corvino, A. Gamatié, and P. Boulet. Architecture exploration for efficient data transfer and storage in data-parallel applications. In *EuroPar'10: Proceedings of the 16th international Euro-Par conference on Parallel processing: Part I*, pages 101–116, Ischia, Italy, 2010. Springer-Verlag. 69

[33] M. Crepaldi, M. R. Casu, M. Graziano, and M. Zamboni. An effective AMS top-down methodology applied to the design of a mixed-signal UWB system-on-chip. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 1424–1429, Nice, France, 2007. EDA Consortium. 22

[34] J. P. David and E. Bergeron. A Step towards Intelligent Translation from High-Level Design to RTL. In *IWSOC '04: Proceedings of the System-on-Chip for Real-Time Applications, 4th IEEE International Workshop*, pages 183–188, Washington, DC, USA, 2004. IEEE Computer Society. 18

[35] G. D.D., S. Abdi, A. Gerstlauer, and G. Schirner. *Embedded System Design: Modeling, Synthesis and Verification*. Springer Publishing Company, Incorporated, 2009. 23

[36] F. de Dinechin. *Systèmes structurés d'équations récurrentes : mise en œuvre dans le langage Alpha et applications*. Thèse de doctorat, université de Rennes I, 1997. MMAlpha available at: http://www.irisa.fr/cosi/ALPHA/mmalpha_english.html,. 31

[37] A. Demeure and Y. Del Gallo. An array approach for signal processing design. In *Sophia-Antipolis conference on Micro-Electronics (SAME'98), System-on-Chip Session, France*, October 1998. 78

[38] A. Demeure, A. Lafage, E. Boutillon, D. Rozzonelli, J. Dufourd, and J. Marro. Array-OL : Proposition d'un formalisme tableau pour le traitement de signal multi-dimensionnel. In *Colloque GRETSI sur le Traitement du Signal et de l'Image, Juan-Les-Pins, France*, September 1995. 27

[39] N. A. V. Doan, F. Robert, Y. D. Smet, and D. Milojevic. MCDA-based methodology for efficient 3D-design space exploration and decision. pages 76–83, September 2010. 16

[40] A. Donlin. Transaction level modeling: flows and use models. In *CODES+ISSS '04: Proceedings of the 2nd IEEE/ACM/IFIP international conference on Hardware/software codesign and system synthesis*, pages 75–80, Stockholm, Sweden, 2004. ACM. 62

[41] Eclipse. Eclipse Modeling Framework. http://www.eclipse.org/emf. 67

[42] B. A. Faten. *Étude et optimisation de l'interaction processeurs-architectures reconfigurables dynamiquement*. PhD thesis, Université Rennes 1, 10 2009. 36

[43] K. Flautner. *Automatic monitoring for interactive performance and power reduction*. PhD thesis, Ann Arbor, MI, USA, 2001. AAI3000950. 43

[44] M. Flynn. Some Computer Organizations and Their Effectiveness. *IEEE Trans. Comput.*, C-21:948+, 1972. 37

[45] J. Forget, F. Boniol, D. Lesens, and C. Pagetti. Implementing Multi-Periodic Critical Systems: from Design to Code Generation. In *FMA*, pages 34–48, 2009. 48

[46] Freescale Semiconductor. Mc1323x 2.4 ghz ieee 802.15.4/zigbee system-on-chip solution. http://cache.freescale.com/files/wireless_comm/doc/fact_sheet/MC1323XFS.pdf?fpsp=1, 2010. 14

[47] Freescale Semiconductor Inc. Freescale semiconductor inc. http://www.frerescale.com. 14

[48] D. D. Gajski and R. H. Kuhn. New vlsi tools. *Computer*, 16(12):11–14, 1983. 19

[49] A. Gamatié. *Designing Embedded Systems with the SIGNAL Programming Language: Synchronous, Reactive Specification*. Springer - New York, New York, USA, 2010. 35

[50] A. Gamatié, S. L. Beux, E. Piel, R. B. Atitallah, A. Etien, P. Marquet, and J.-L. Dekeyser. A Model Driven Design Framework for Massively Parallel Embedded Systems. *TECS: ACM Transactions on Embedded Computing Systems l' ACM (To appear), preliminary version at http://hal.inria.fr/inria-00311115/*, 2010. 61

[51] A. Gamatié, T. Gautier, P. L. Guernic, and J.-P. Talpin. Polychronous design of embedded real-time applications. *ACM Trans. Softw. Eng. Methodol.*, 16, April 2007. 114

[52] J.-L. Gaudiot, T. DeBoni, J. Feo, W. Böhm, W. Najjar, and P. Miller. The Sisal Model of Functional Programming and its Implementation. In *PAS'97: Proceedings of the 2nd AIZU International Symposium on Parallel Algorithms/Architecture Synthesis*, pages 112–, Washington, DC, USA, 1997. IEEE Computer Society. 26

[53] A. Gill. *Introduction to the theory of finite-state machines*. McGraw-Hill electronic sciences series. McGraw-Hill, New York, 1962. 26, 28

[54] C. Glitia. *OPTIMISATION DES APPLICATIONS DE TRAITEMENT SYSTÉMATIQUE INTENSIVES SUR SYSTEMS-ON-CHIP*. PhD thesis, University of Lille1, november 2009. 56

[55] C. Glitia, J. Deantoni, and F. Mallet. Logical time at work: capturing data dependencies and platform constraints. In *FDL'10: Forum for Design Languages, Proceedings of the 2010 Forum on specification & Design Languages*, pages 240–246, Southampton Royaume-Uni, 09 2010. Electronic Chips & Systems design Initiative (ECSI). 45

[56] P. Grosse, Y. Durand, and P. Feautrier. Methods for power optimization in SOC-based data flow systems. *ACM Trans. Des. Autom. Electron. Syst.*, 14(3):1–20, 2009. 51

[57] H. Yu. *A MARTE based reactive model for data-parallel intensive processing: Transformation toward the synchronous model*. PhD thesis, USTL, 2008. 62, 68

[58] A. Habibi and S. Tahar. A survey on system-on-a-chip design. *Proceedings 3rd IEEE International Workshop on System-on-Chip for Real-Time Applications*, pages 212–215, july 2003. 4

[59] S.-I. Han, S.-I. Chae, L. Brisolara, L. Carro, K. Popovici, X. Guerin, A. A. Jerraya, K. Huang, L. Li, and X. Yan. Simulink®-based heterogeneous multiprocessor SoC design flow for mixed hardware/software refinement and simulation. *Integr. VLSI J.*, 42(2):227–245, 2009. 19

[60] IBM. PowerPC 405 CPU Core. Technical report, september 2006. 170

[61] Ieee. *Std 1076-2000: IEEE Standard VHDL Language Reference Manual*. IEEE, 2000. 21

[62] R. Jejurikar, C. Pereira, and R. Gupta. Leakage aware dynamic voltage scaling for real-time embedded systems. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 275–280, San Diego, CA, USA, 2004. ACM. 36

[63] M.-J. Jung, Y. R. Seong, and C.-H. Lee. Optimal RM scheduling for simply periodic tasks on uniform multiprocessors. In *ICHIT '09: Proceedings of the 2009 International Conference on Hybrid Information Technology*, pages 383–389, Daejeon, Korea, 2009. ACM. 49

[64] G. Kahn. The Semantics of a Simple Language for Parallel Programming. In J. L. Rosenfeld, editor, *Information Processing '74: Proceedings of the IFIP Congress*, pages 471–475, New York, USA, 1974. North-Holland. 26, 28, 29

[65] T. Kangas, P. Kukkala, H. Orsila, E. Salminen, M. Hännikäinen, T. D. Hämäläinen, J. Riihimäki, and K. Kuusilinna. UML-based multiprocessor SoC design framework. *ACM Trans. Embed. Comput. Syst.*, 5:281–320, May 2006. 28

[66] L. Karam, I. Alkamal, A. Gatherer, G. Frantz, D. Anderson, and B. Evans. Trends in multicore DSP platforms. *IEEE Signal Processing Magazine*, 26(6):38–49, November 2009. 39

[67] J. Keinert, M. Streub&uhorbar;hr, T. Schlichter, J. Falk, J. Gladigau, C. Haubelt, J. Teich, and M. Meredith. SystemCoDesigner—an automatic ESL synthesis approach by design space exploration and behavioral synthesis for streaming applications. *ACM Trans. Des. Autom. Electron. Syst.*, 14(1):1–23, 2009. Available at: http://www12.informatik.uni-erlangen.de/research/scd/. 22, 31

[68] C.-C. Kuo, Y.-C. Wang, and C.-N. J. Liu. An efficient bottom-up extraction approach to build accurate PLL behavioral models for SoC designs. In *GLSVLSI '05: Proceedings of the 15th ACM Great Lakes symposium on VLSI*, pages 286–290, Chicago, Illinois, USA, 2005. ACM. 22

[69] L. Lamport. Time, clocks, and the ordering of events in a distributed system. *Commun. ACM*, 21(7):558–565, July 1978. 35, 92

[70] E. Lee, C. Hylands, J. Janneck, J. Davis II, J. Liu, X. Liu, S. Neuendorffer, S. Sachs, M. Stewart, K. Vissers, and P. Whitaker. Overview of the Ptolemy Project. Technical Report UCB/ERL M01/11, EECS Department, University of California, Berkeley, 2001. Available at ptolemy.eecs.berkeley.edu/publications/papers/01/overview. 28

[71] E. A. Lee and David. Static Scheduling of Synchronous Data Flow Programs for Digital Signal Processing. *IEEE Transactions on Computers*, 36:24–35, 1987. 28

[72] E. A. Lee and D. G. Messerschmitt. Synchronous Data Flow: Describing Signal Processing Algorithm for Parallel Computation. In *COMPCON*, pages 310–315, 1987. 26

[73] C. L. Liu and J. W. Layland. Scheduling Algorithms for Multiprogramming in a Hard-Real-Time Environment. *J. ACM*, 20:46–61, January 1973. 41

[74] F. Mallet. Clock constraint specification language: specifying clock constraints with UML/MARTE. *Innovations in Systems and Software Engineering*, 4(3):309–314, October 2008. 78

[75] G. Martin and S. Leibson. Commercial Configurable Processors and the MESCAL Approach. In M. Gries and K. Keutzer, editors, *Building ASIPS: The Mescal Methodology*, pages 281–310. Springer US, 2005. 29

[76] MathWorks. MATLAB–The Language Of Technical Computing, september 2010. 20

[77] J. L. Medina, M. G. Harbour, and J. M. Drake. The "UML Profile for Schedulability, Performance and Time" in the Schedulability Analysis and Modeling of Real-Time Distributed Systems. In *SIVOES-SPT Workshop*, Toronto, Canada, May 2004. 24

[78] A. MEENA. *Allocation, Assignment and Scheduling for Multi-processor System on Chip*. PhD thesis, USTL, 2006. 69

[79] H. Metivier, J.-P. Talpin, T. Gautier, and P. L. Guernic. *Distributed Embedded Systems: Design, Middleware and Resources*, volume 271/2008, chapter Analysis of Periodic Clock Relations in Polychronous Systems. Springer Boston, 2008. 45

[80] D. Milojevic, T. Carlson, K. Croes, R. Radojcic, D. F. Ragett, D. Seynhaeve, F. Angiolini, G. V. der Plas, and P. Marchal. Automated Pathfinding tool chain for 3D-stacked integrated circuits: Practical case study. In *3DIC*, pages 1–6, 2009. 16

[81] P. Mishra, A. Shrivastava, and N. Dutt. Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable SOCs. In *DAC '04: Proceedings of the 41st annual Design Automation Conference*, pages 626–658, San Diego, CA, USA, 2004. ACM. 22

[82] B. C. Mochocki, K. Lahiri, S. Cadambi, and X. S. Hu. Signature-based workload estimation for mobile 3D graphics. In *DAC'06: Proceedings of the 43rd annual Design Automation Conference*, pages 592–597, San Francisco, CA, USA, 2006. ACM. 123

[83] J. E. Nielsen and K. S. Knudsen. MoVES - A tool for Modelling and Verification of Embedded Systems, 2007. Master Thesis, Supervised by Associate Professor Michael R. Hansen and Professor Jan Madsen,. 93

[84] Object Management Group. UML Profile for System on a Chip (SoC), 2001. http://www.omgmarte.org. 24

[85] Object Management Group. Unified Modeling Language, Version 2.2. 2009. http://www.omgmarte.org/Specification.htm. 20, 24, 67

[86] Object Management Group Inc. Final Adopted OMG SysML Specification. http://www.omg.org/cgi-bin/doc?ptc/06-0504, mai 2006. 24

[87] Object Management Group Inc. UML 2 Infrastructure (Final Adopted Specifcation). http://www.omg.org/spec/UML/2.1.2/Infrastructure/PDF/, Nov. 2007. 87

[88] OCaml Homepage. The OCaml language. http://caml.inria.fr/. 130

[89] OMG. Modeling and Analysis of Real-time and Embedded systems(MARTE). http://www.omgmarte.org/. 6, 24, 56, 59

[90] OMG. Portal of the Model Driven Engineering Community, 2007. http://www.planetmde.org. 66

[91] E. Piel. *Ordonnancement de systèmes parallèles temps-réel*. PhD thesis, Université des Sciences et Technologies de Lille (USTL), 2007. 61

[92] A. D. Pimentel. The Artemis workbench for system-level performance evaluation of embedded systems. *IJES*, 3(3):181–196, 2008. http://www.ce.et.tudelft.nl/artemis. 29

[93] D. Potop-Butucaru, R. D. Simone, and J.-P. Talpin. The syncrhonous hypothesis and synchronous languages. In *Embedded Systems Handbook,R. Zurawski*. CRC Press and ed., 2005. 28, 78

[94] I. Quadri. *MARTE based model driven design methodology for targeting dynamically reconfigurable FPGA based SoCs*. PhD thesis, University of Lille1, 2010. 21, 62, 69

[95] I. R. Quadri, S. Meftali, and J.-L. Dekeyser. High level modeling of dynamic reconfigurable FPGAs. *Int. J. Reconfig. Comput.*, 2009:1–15, 2009. 19

[96] E. Riccobene, P. Scandurra, S. Bocchio, A. Rosti, L. Lavazza, and L. Mantellini. SystemC/C-based model-driven design for embedded systems. volume 8, pages 1–37, New York, NY, USA, 2009. ACM. 24

[97] E. Riccobene, P. Scandurra, A. Rosti, and S. Bocchio. A model-driven design environment for embedded systems. In *DAC '06: Proceedings of the 43rd annual Design Automation Conference*, pages 915–918, San Francisco, CA, USA, 2006. ACM. 19

[98] A. Richard, C. Hernalsteens, and F. Robert. Development and Validation of Nessie : a multi-criteria performance estimation tool for SoC. In *PRIME'09*, Cork, Ireland, 2009. IEEE. 29

[99] J. Ricken. *Testing the SOA domain model for a model driven and process-oriented SOA implementation*. PhD thesis, University of Namur, Computer Science Department, 2009. 22

[100] L. Rioux, T. Saunier, S. Gérard, A. Radermacher, R. De Simone, T. Gautier, Y. Sorel, J. Forget, J.-L. Dekeyser, A. Cuccuru, C. Dumoulin, and C. André. MARTE: A New OMG Profile RFP for the Modeling and Analysis of Real-Time Embedded Systems. In *DAC 2005: Workshop UML for SoC Design, UML-SoC'05*, Anaheim CA, USA, June 2005. 56

[101] B. Ristau, T. Limberg, O. Arnold, and G. Fettweis. Dimensioning heterogeneous mpsocs via parallelism analysis. In *DATE '09: Proceedings of the Conference on Design, Automation and Test in Europe*, pages 554–557, Nice, France, 2009. European Design and Automation Association. Available at: http://portal.acm.org/citation.cfm?id=1874620.1874755. 52

[102] K. Sakuma, P. S. Andry, C. K. Tsang, S. L. Wright, B. Dang, C. S. Patel, B. C. Webb, J. Maria, E. J. Sprogis, S. K. Kang, R. J. Polastre, R. R. Horton, and J. U. Knickerbocker. 3D chip-stacking technology with through-silicon vias and low-volume lead-free interconnections. *IBM J. Res. Dev.*, 52:611–622, November 2008. 16

[103] A. Sangiovanni-Vincentelli, L. Carloni, F. De Bernardinis, and M. Sgroi. Benefits and challenges for platform-based design. In *DAC'04: Proceedings of the 41st annual Design Automation Conference*, pages 409–414, San Diego, CA, USA, 2004. ACM. 23

[104] Sanjay Krishnan. Cost Realities in Bringing a Chip to Market. Available at: http://www.gsaglobal.org/forum/2010/3/articles_maxim.asp. 17

[105] S.-B. Scholz. Single Assignment C: efficient support for high-level array operations in a functional setting. *J. Funct. Program.*, 13:1005–1059, November 2003. 27

[106] C. Seo, G. Edwards, D. Popescu, S. Malek, and N. Medvidovic. A framework for estimating the energy consumption induced by a distributed system's architectural style. In *SAVCBS '09: Proceedings of the 8th international workshop on Specification and verification of component-based systems*, pages 27–34, Amsterdam, The Netherlands, 2009. ACM. 49

[107] E. Sicard and S. B. Dhia. Dessin et simulation de fonctions de base en cmos 90 nm. *j3eA*, 4(23):4, 2005. 18

[108] A. Siebenborn, O. Bringmann, and W. Rosenstiel. Communication Analysis for System-On-Chip Design. In *DATE '04: Proceedings of the conference on Design, automation and test in Europe*, page 10648, Washington, DC, USA, 2004. IEEE Computer Society. 45

[109] G. S. Silveira, A. V. Brito, and E. U. K. Melcher. Functional verification of power gate design in SystemC RTL. In *SBCCI '09: Proceedings of the 22nd Annual Symposium on Integrated Circuits and System Design: Chip on the Dunes*, pages 52:1–52:5, Natal, Brazil, 2009. ACM. 4

[110] Simulink. Simulation and Model-Based Design, 09 2010. http://www.mathworks.com/products/simulink. 28

[111] I. Smarandache. *Transformations affines d'horloges: application au codesign de systèmes temps rel en utilisant les langages SIGNAL et ALPHA*. PhD thesis, Universit de Rennes 1, Rennes, France, October 1998. 41, 45, 112, 113

[112] SoCLib. SoCLib: An open platform for modelling and simulation of multi-processors system on chip, 2009. https://www.soclib.fr/trac/dev/wiki. 61

[113] Solid-State Circuits Newsletter, IEEE. Hugo De Man Awarded For Leadership In Integrated Circuit Design. 23

[114] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2000. 20

[115] C. Studio. Cofluent design. Available at: http://www.cofluentdesign.com/, 2008. 28

[116] T. Mens and P. Van Gorp. A taxonomy of model transformation. In *GraMoT'05: Proceedings of the International Workshop on Graph and Model Transformation*, pages 125–142, 2006. 67

[117] J. Taillard. *Une approche orientée modèle pour la parallélisation d'un code de calcul éléments finis*. PhD thesis, Lille, France, 2009. 62, 69

[118] J. S. Tesnim Abdellatif, Jacques Combaz. Model-Based Implementation of Real-Time Applications. Technical Report TR-2010-14, Verimag Research Report, 2010. 48

[119] TEXAS INSTRUMENT Corporation. Realizing full multicore entitlement, February 2010. Available at: http://focus.ti.com/lit/wp/spry133/spry133.pdf. 15

[120] TEXAS INSTRUMENTS Inc. Texas instruments incs. http://www.ti.com/. 15

[121] W. Thies, M. Karczmarek, M. Gordon, D. Maze, J. Wong, H. Hoffman, M. Brown, and S. Amarasinghe. Streamit: A compiler for streaming applications. MIT/LCS Technical Memo MIT/LCS Technical Memo LCS-TM-622, Massachusetts Institute of Technology, Cambridge, MA, December 2001. 27

[122] D. Thomas and P. Moorby. *The Verilog Hardware Description Language*. Springer Publishing Company, Incorporated, 2008. 21

[123] M. Thompson, H. Nikolov, T. Stefanov, A. D. Pimentel, C. Erbas, S. Polstra, and E. F. Deprettere. A framework for rapid system-level exploration, synthesis, and programming of multimedia MP-SoCs. In *CODES+ISSS '07: Proceedings of the 5th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 9–14, Salzburg, Austria, 2007. ACM. 4

[124] P. Urard, A. Maalej, R. Guizzetti, and N. Chawla. Leveraging sequential equivalence checking to enable system-level to RTL flows. In *DAC '08: Proceedings of the 45th annual Design Automation Conference*, pages 816–821, Anaheim, California, 2008. ACM. 18

[125] B. Vallet and B. Lévy. Spectral Geometry Processing with Manifold Harmonics. *Computer Graphics Forum (Proceedings Eurographics)*, 2008. 3

[126] A. Viehl, M. Pressler, and O. Bringmann. Bottom-up performance analysis considering time slice based software scheduling at system level. In *CODES+ISSS '09: Proceedings of the 7th IEEE/ACM international conference on Hardware/software codesign and system synthesis*, pages 423–432, Grenoble, France, 2009. ACM. 48

[127] G. K. Wallace. The JPEG still picture compression standard. *Commun. ACM*, 34(4):30–44, 1991. 144

[128] R. Watanabe, M. Kondo, M. Imai, H. Nakamura, and T. Nanya. Task scheduling under performance constraints for reducing the energy consumption of the GALS multi-processor SoC. In *DATE '07: Proceedings of the conference on Design, automation and test in Europe*, pages 797–802, Nice, France, 2007. EDA Consortium. 49

[129] Wikipedia. Arrivé avant. http://fr.wikipedia.org/wiki/Arriv 35

[130] Wikipedia. JPEG. http://fr.wikipedia.org/wiki/JPEG. 144

[131] Wikipedia. System on Chip. http://fr.wikipedia.org/wiki/System_on_Chip. 14

[132] D. Wilde. The alpha language. Technical Report 827, IRISA - INRIA, Rennes, 1994. Available at www.irisa.fr/bibli/publi/pi/1994/827/827.html. 26

[133] W. Wolf, A. A. Jerraya, and G. Martin. Multiprocessor System-on-Chip (MPSoC) Technology. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 27(10):1701–1713, October 2008. 15

[134] Z. Xiong, S. Li, and J. Chen. Hardware/software co-design using hierarchical platform-based design method. In *ASP-DAC'05: Proceedings of the 2005 Asia and South Pacific Design Automation Conference*, pages 1309–1312, Shanghai, China, 2005. ACM. 4

[135] H. Yang and S. Ha. Pipelined data parallel task mapping/scheduling technique for mpsoc. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '09, pages 69–74, Nice, France, 2009. European Design and Automation Association. Available at: http://portal.acm.org/citation.cfm?id=1874620.1874638. 52

[136] H. Yu, A. Gamatié, E. Rutten, and J.-L. Dekeyser. Safe design of high-performance embedded systems in a mde framework. *Innovations in Systems and Software Engineering (ISSE)*, 4(3):215–222, 2008. 68

[137] W. Zhu, V. C. Sreedhar, Z. Hu, and G. R. Gao. Synchronization state buffer: supporting efficient fine-grain synchronization on many-core architectures. In *ISCA '07: Proceedings of the 34th annual international symposium on Computer architecture*, pages 35–45, San Diego, California, USA, 2007. ACM. 45

[138] Y. Zhu and F. Mueller. Exploiting synchronous and asynchronous DVS for feedback EDF scheduling on an embedded platform. *ACM Trans. Embed. Comput. Syst.*, 7(1):1–26, 2007. 48, 49

Curriculum Vitæ

Publication [6]

Journaux [1]

1. Adolf Abdallah, Abdoulaye Gamatié and Jean-Luc Dekeyser "**Modélisation UML/MARTE de SoC et analyse temporelle basée sur l'approche synchrone**", *Technique et science informatiques* (TSI), volume 30, France, 2010.

Conférences Internationales À Comité De Lecture Et Actes [4]

1. Adolf Abdallah, Abdoulaye Gamatié and Jean-Luc Dekeyser "**Correct and Energy-Efficient Design of SoCs: the H.264 Encoder Case Study**", *International Symposium on System-on-Chip* (SoC'2010), Tampere, Finland, September 2010. IEEE Press (BEST PAPER AWARD).

2. Adolf Abdallah, "**High-Level SoC Specification Towards a Design Space Exploration : Downscaler Case Study**", *EuroDocInfo10*, Valenciennes, France, January 2010.

3. Adolf Abdallah, Abdoulaye Gamatié and Jean-Luc Dekeyser"**Model-Driven Design of Embedded Multimedia Applications on SoCs**", *12th Euromicro Conference on Digital System Design*(DSD'2009), Patras, Greece, August 2009. IEEE Press.

4. Adolf Abdallah, Abdoulaye Gamatié and Jean-Luc Dekeyser "**MARTE-based Design of a Multimedia Application and Formal Analysis**", *Forum on specification and Design Languages* (FDL'08), Stuttgart, Germany, September 2008. IEEE Press.

Conférences nationales À Comité De Lecture Et Actes [2]

1. Adolf Abdallah, Abdoulaye Gamatié and Jean-Luc Dekeyser "**Modélisation UML/MARTE de SoC et analyse temporelle basée sur l'approche synchrone**", *Symposium en Architecture de Machines* (SYMPA'13), Toulouse, France, Septembre 2009.

2. Adolf Abdallah "**Modélisation UML/MARTE et Analyse Temporelle pour une Exploration Architecturale des Systemes-sur-Puce**, *ParisTech* (ETR'09), Paris, France, Septembre 2009.

Prix et distinctions [1]

1. *Best Paper Award - International Symposium on System-on-Chip* (SoC'2010) - obtenu pour un article présenté dans le cadre d'une conférence internationale en 2010 sur les systèmes-sur-puce à Tampere, Finlande (voir la rubrique Publications).

Enseignement [2]

1. **Moniteur en Informatique à l'université Lille 1**, UFR d'IEEA du premier octobre 2007 jusqu'à fin septembre 2010.

2. **Attaché temporaire à l'enseignement supérieur (A.T.E.R) à l'université Lille 1**, UFR d'IEEA du premier octobre 2010 jusqu'à fin décembre 2010.

École d'été [2]

1. **Real Time Summer School 2009** *ParisTech*, Paris, France, August 2009.
2. **The 2008 Lectures in Computer Science: Embedded networked Systems: Theory and Applications**, Heraklion, Crete, Greece, July 2008.

ABSTRACT

Clock Based SoC Design, Towards a Design Space Exploration in MARTE

Modern high-performance embedded applications as found in multimedia, biomedical signal processing or biometric data processing are increasingly complex and resource-demanding. The quest for the ultimate execution performance on single processor chips is unfortunately a dead-end. Instead, the promising solution is Multi-Processor System-on-Chip (MPSoC). However, the design of MPSoCs for high-performance embedded applications is a very difficult task due to a number of crucial constraints to meet: functionality correctness, temporal performance, energy efficiency and optimized memory size. Among the necessary ingredients for a successful design, we mention first the need of expressive enough programming models for describing the potential parallelism inherent to target applications. Second, we need ways avoiding tedious explorations of best architecture configurations for system execution (e.g. processors type and frequency, memory footprint), especially for complex data-intensive applications mapped on massively parallel architectures. Third, several abstraction levels must be taken into account to better address design complexity. Considering a single simulation level at which all implementation details are considered yields very accurate results, but is more time consuming and tedious, even sometimes impossible due to system complexity. For this reason, starting the design process at a high abstraction level, where only key information about the system are described makes it easier to take early decisions about the configuration choices at a very low cost.

In order to deal with the above challenges concerning the design of high-performance applications on MPSoCs, we propose in this thesis to use the MARTE/UML profile for the modeling of system functionality, execution architectures and allocation of both parts. The MARTE profile is expressive enough to describe high performance applications (e.g. RSM and GCM packages and the CCSL language) and MPSoC architectures (e.g. HRM package). For the verification of the system and the design space exploration, we define an abstraction of the resulting model with abstract clocks, inspired by those of synchronous reactive languages. The traces associated with such clocks capture the behavior of a system by representing the activity of processing units, i.e. processors, when achieving functionality. An analysis technique, also inspired by the synchronous approach, is also defined. It enables to verify functional constraints: data dependencies induced by a MARTE model, activation rate constraints between components. In addition, it allows us to deal with non functional properties: execution time, deadline preservation, energy consumption. These properties are directly related to the number of processors involved in system execution as well as their associated frequency values. From an overall viewpoint, the main contribution of this thesis is the definition an abstract clock-based framework that aims to facilitate MPSoC design space exploration at a high abstraction level. It has been made concrete within an environment, referred to as GASPARD, dedicated to the codesign of high-performance embedded systems. Our solution is validated on a case study consisting of a JPEG encoder, with very promising results.

Keywords: Model-Driven Engineering, MARTE, UML, SoC Co-Design, Gaspard2, Embedded Systems, Desgin Space Exploration, validation, Synchronous, Simulation.

RÉSUMÉ

Conception de SoC à Base d'Horloges Abstraites : Vers l'Exploration d'Architectures en MARTE

Les applications modernes embarquées à hautes performances telles que l'on trouve dans les domaines du multimédia, du traitement de signaux biomédicaux et du traitement de données biométriques, sont de plus en plus complexes et exigeantes en termes de ressources. L'augmentation des performances de puces, contenant un seul processeur, n'est plus une solution adoptée. Par conséquent, une solution prometteuse est les systèmes-sur-puce multiprocesseurs (MPSoC). Cependant, la conception de MPSoC dédiés aux traitements d'applications hautes performances est un travail très difficile en raison d'un certain nombre de contraintes à assurer : la correction fonctionnelle, les performances temporelles, l'efficacité énergétique et la taille optimisée de mémoire. Parmi les ingrédients nécessaires pour une construction correcte et optimisée, nous citons premièrement le besoin de modèles de programmation assez expressifs pour décrire le parallélisme potentiel inhérent des applications cibles. Deuxièmement, nous devons trouver des moyens pour éviter des explorations fastidieuses afin de trouver la meilleure configuration d'architecture pour l'exécution d'une application (par exemple le type et la fréquence des processeurs, l'empreinte mémoire), en particulier pour des applications complexes manipulant de grandes quantités de données et exécutées par des architectures massivement parallèles. Troisièmement, plusieurs niveaux d'abstraction doivent être pris en compte afin de mieux traiter la complexité de la conception. En considérant un niveau de simulation unique où tous les détails d'implémentations sont considérés, les résultats obtenus auront une grande précision. Cependant, cette technique de simulation est longue et fastidieuse, voir parfois impossible en raison de la grande complexité de systèmes. Pour cette raison, le commencement du processus de conception, à un haut niveau d'abstraction, où seulement des informations essentielles de systèmes sont décrites, rend plus facile, et à un très faible coût, la prise de décision sur des choix de configuration.

Afin de relever les défis mentionnés ci-dessus concernant la conception des applications MPSoC à haute performance, nous proposons dans le cadre de cette thèse, l'utilisation du profil UML/Marte pour la modélisation de fonctionnalité, d'architectures et d'associations des deux dernières. Ce profil est suffisamment expressif pour décrire des applications hautes performances (par exemple les paquetages RSM et GCM et le langage CCSL) et des architectures MPSoC massivement parallèles (par exemple le paquetage HRM). Pour l'analyse et la vérification de systèmes et l'exploration de l'espace de conception, nous définissons une abstraction de modèles obtenue via des horloges abstraites, inspirées de ceux des langages réactifs synchrones. Les traces d'horloges abstraites capturent les comportements de systèmes en représentant l'activité des unités de traitements durant l'exécution de fonctionnalités. Une technique d'analyse, également inspirée de l'approche synchrone, est définie. Cette technique permet de vérifier des contraintes temporelles : dépendances de données induites par un modèle Marte, les taux d'activations entre des composants. En outre, elle permet d'analyser des contraintes non fonctionnelles : estimation de temps d'exécution, le respect des temps d'échéance, estimation de la consommation d'énergie. Ces propriétés sont directement liées au nombre de processeurs impliqués dans l'exécution du système ainsi que la valeur de leurs fréquences associées. D'un point de vue général, la contribution principale de cette thèse est la définition d'un cadre de travail, à base d'horloges abstraites, qui facilite l'exploration de l'espace de conception des MPSoC à un haut niveau d'abstraction. Le travail a été concrétisé dans un environnement, dénommé Gaspard2, dédié à la conception conjointe de systèmes embarqués à hautes performances. Notre solution est validée sur une étude de cas d'un encodeur JPEG, et retournant des résultats prometteurs.

Mots clés: Ingénierie dirigée par les modèles, MARTE, UML, Co-Modélisation de SoC, Gaspard2, Systèmes Embarqués, Exploration de l'Espace de Conception, Validation, Synchrone, Simulation.